Lecture 6

Collision Detection and Proximity Queries (Part 2)

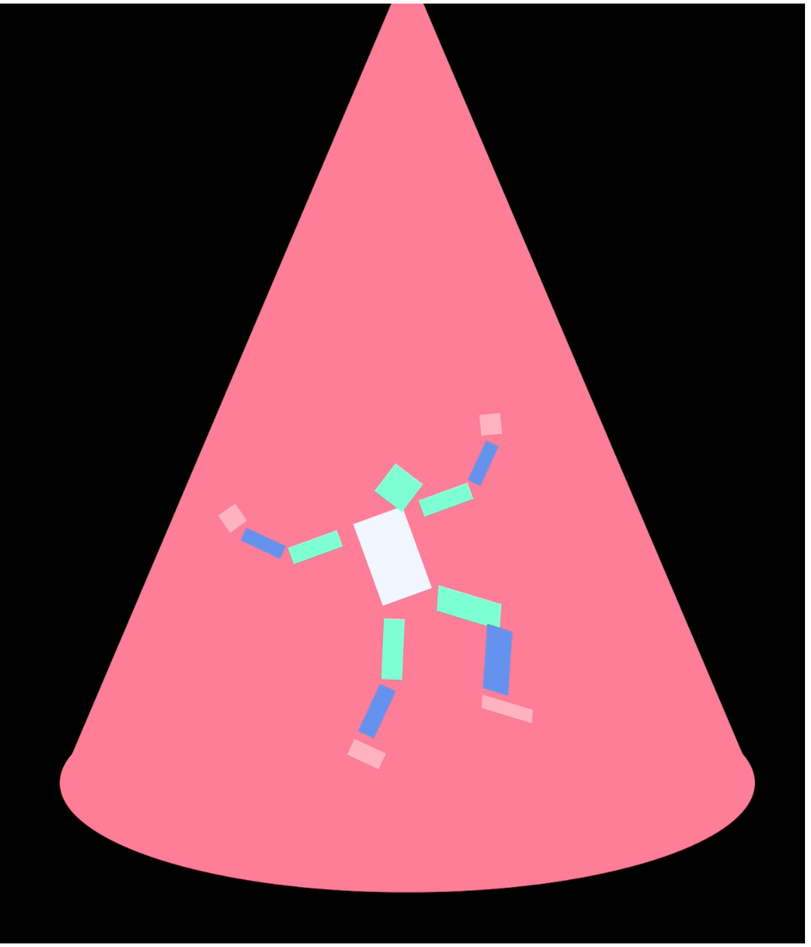
Fundamentals of Computer Graphics

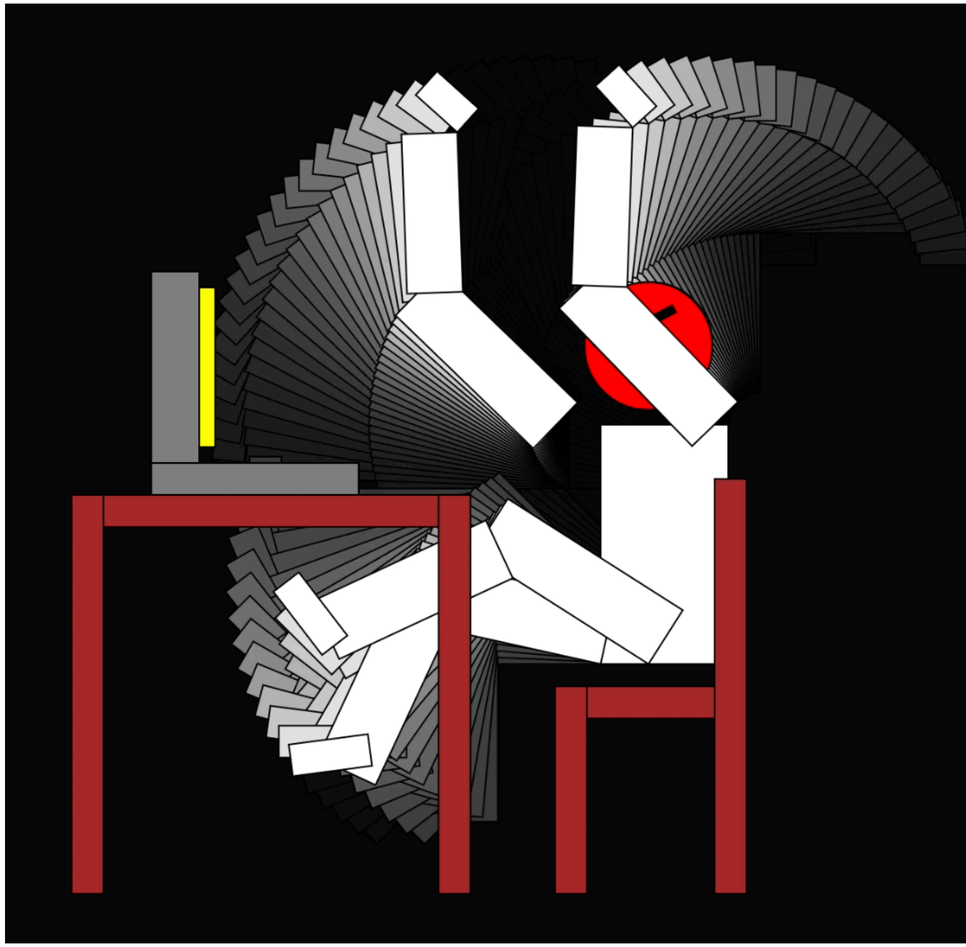
Animation & Simulation

Stanford CS248B, Fall 2023

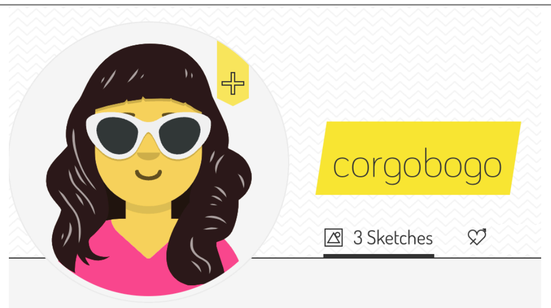
Profs. Karen Liu & Doug James

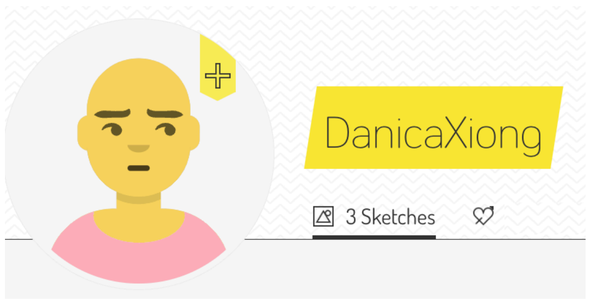
HW1



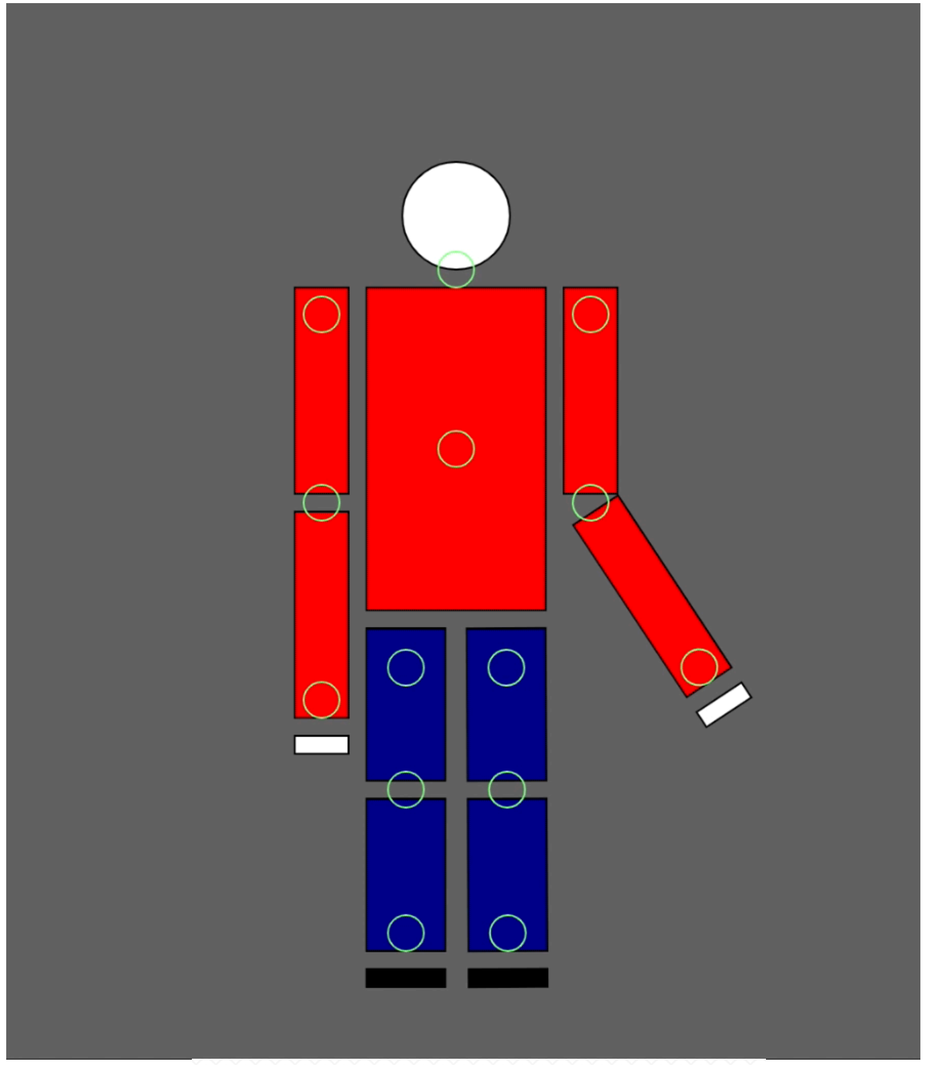



kkdninjia
4 Sketches


corgobogo
3 Sketches


DanicaXiong
3 Sketches

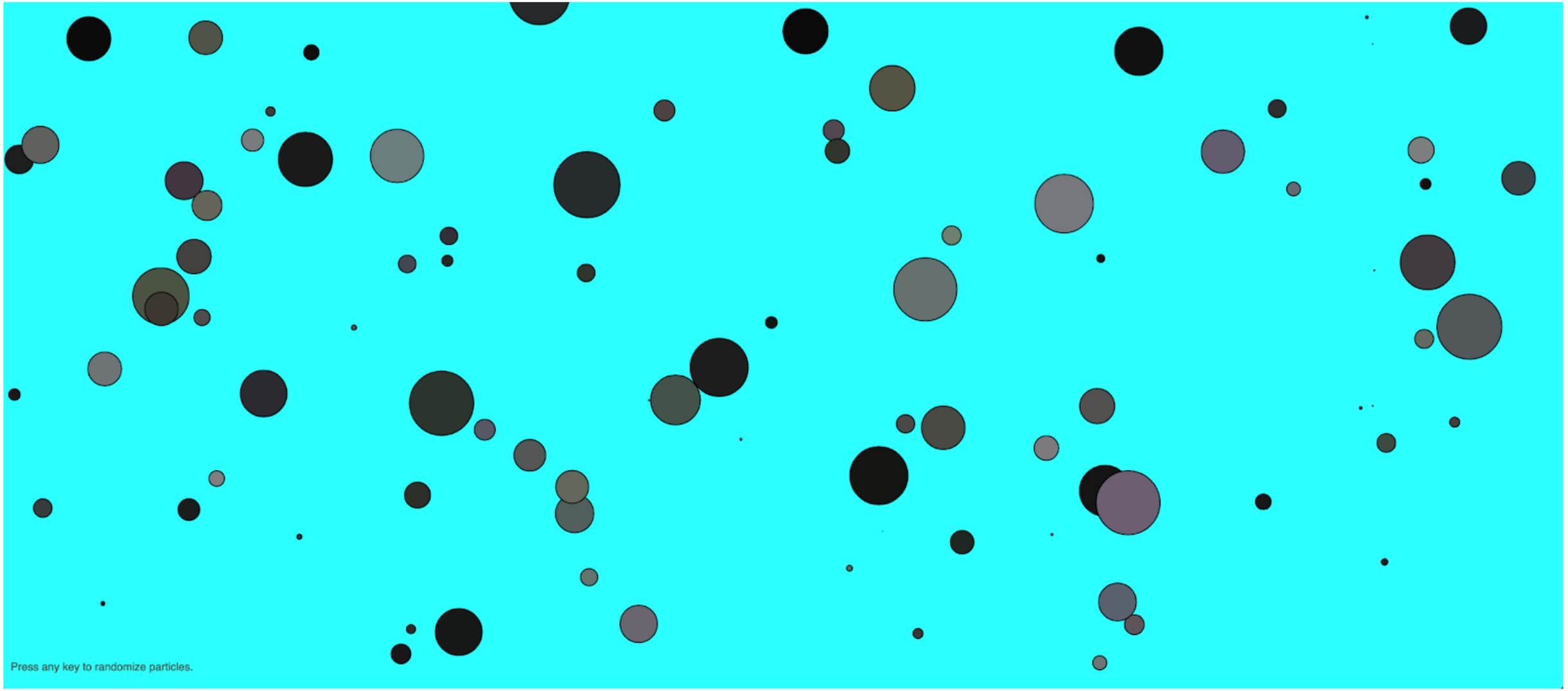
HW1



Broad Phase Culling

Speeding up the broad phase:

Culling pairwise checks for distant pairs



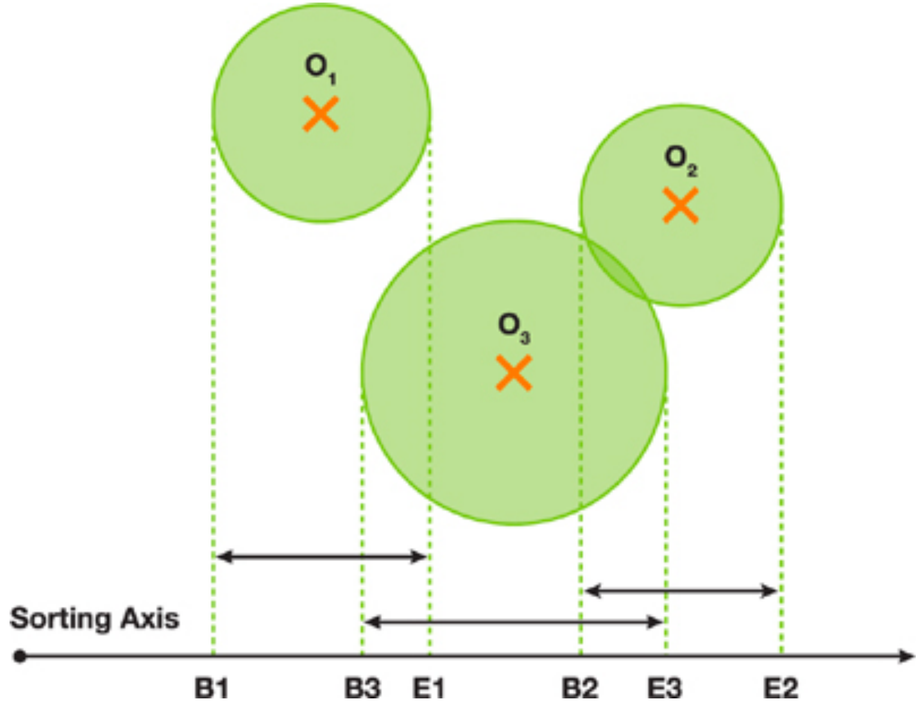
Broad phase computations

Broad Phase:

- Coarse object-level view of simulation, e.g., as bounding volumes
- Determines object pairs that may collide, and culls other pairs
- Usually fast
- Acceleration schemes:
  - Sorting
    - Sweep & Prune
  - Spatial subdivisions
    - Uniform grids; hashing
    - Adaptive grids: quadtree (2D) and octree (3D), kd-trees, H-Grids, etc.
  - Trees:
    - Bounding Volume Hierarchy (BVH)

Sort & Sweep (or Sweep & Prune)

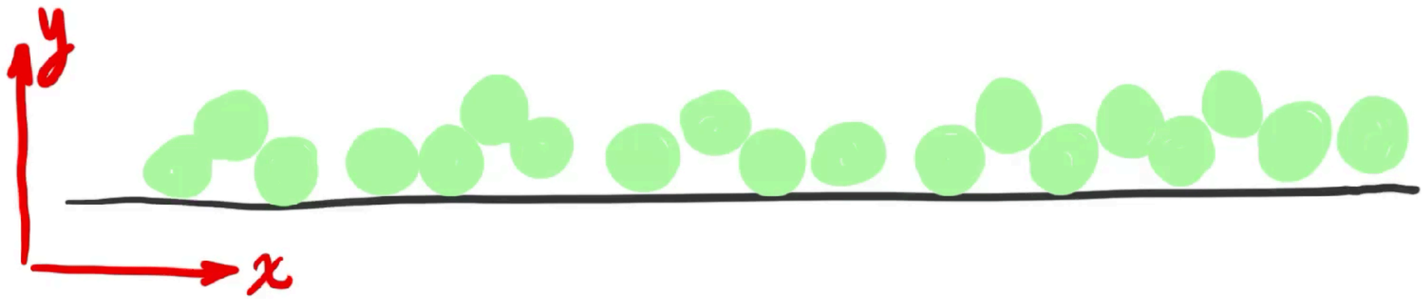
- Sort and Sweep algorithm [Witkin and Baraff 1997]
  - David Baraff Ph.D. thesis [1992]
  - I-COLLIDE [Cohen et al. 1995] (Sweep & Prune)
- Apply 1D overlap tests on sorted interval lists
- Given objects' 1D projections as list of $\{[B_i,E_i]\}_i$ pairs
- Sort all values
  - Exploit temporal coherence for fast re-sorting, e.g., insertion sort
- Initialize active objects to $\varnothing$
- Sweep the list to find overlapping intervals:
  - For each begin marker, B_k:
    - perform CD between k and list of active objects
    - add k to list of active objects
  - For each end marker, E_k, remove k from list of active objects



https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-32-broad-phase-collision-detection-cuda

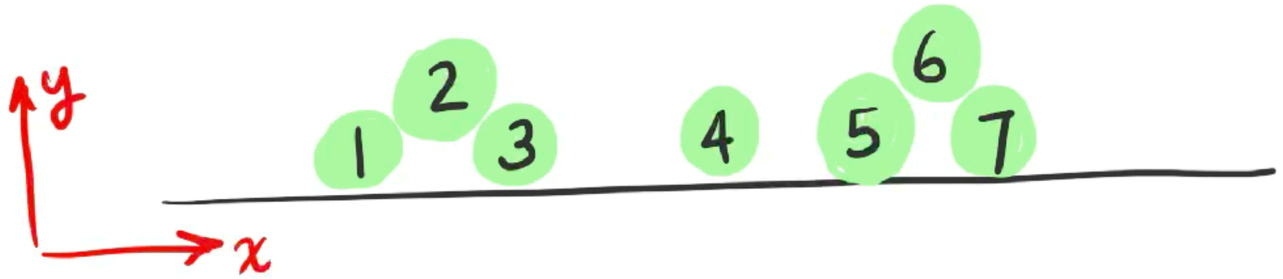
- What can go wrong?

Sorting vs Bucketing



- Problems with sorting:
  - Necessary?
  - O(N lg N) sorting cost
    - Loss of temporal coherence increases re-sorting cost
  - Limited to 1D
    - Can be many false positives (e.g., sorting on y ↑)
    - Can sort in multiple directions and intersect results (how? issues?)

EX: "Sort & Sweep" in multiple directions



SORT ON → x

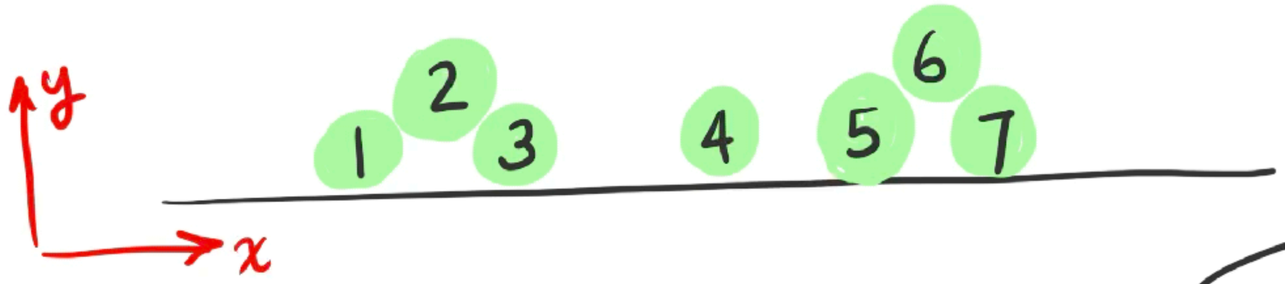
(1,2)

(2,3)

(5,6)

(6,7)

EX: "Sort & Sweep" in multiple directions



SORT ON → x

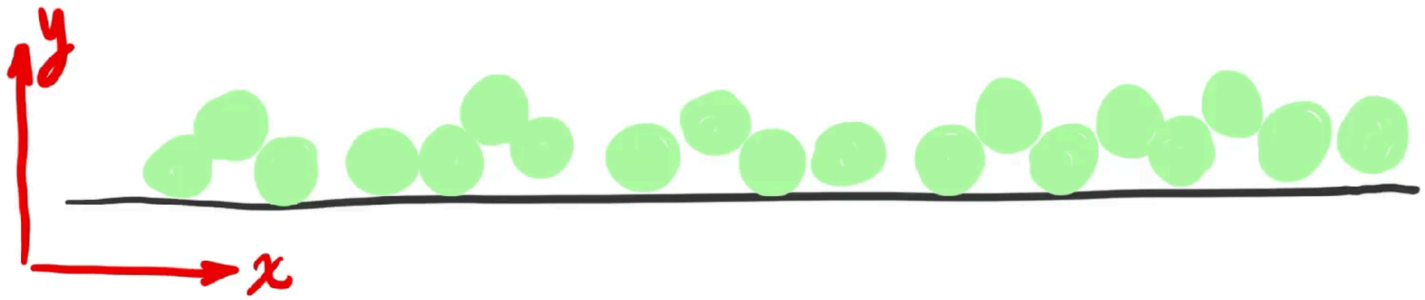
(1,2)

(2,3)

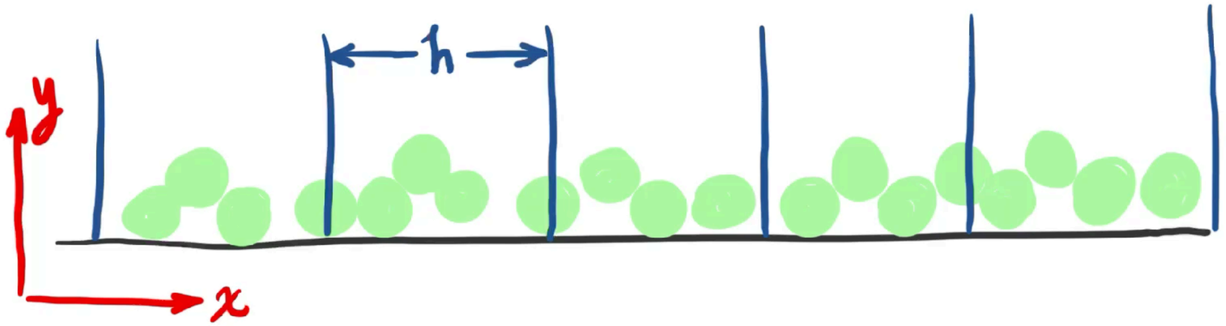
(5,6)

(6,7)

Sorting vs Bucketing



- Problems with sorting:
  - Necessary?
  - O(N lg N) sorting cost
    - Loss of temporal coherence increases re-sorting cost
  - Limited to 1D
    - Can be many false positives (e.g., sorting on y ↑ generates $O(N^2)$ pairs)
    - Can sort in multiple directions and intersect results (how? issues?)



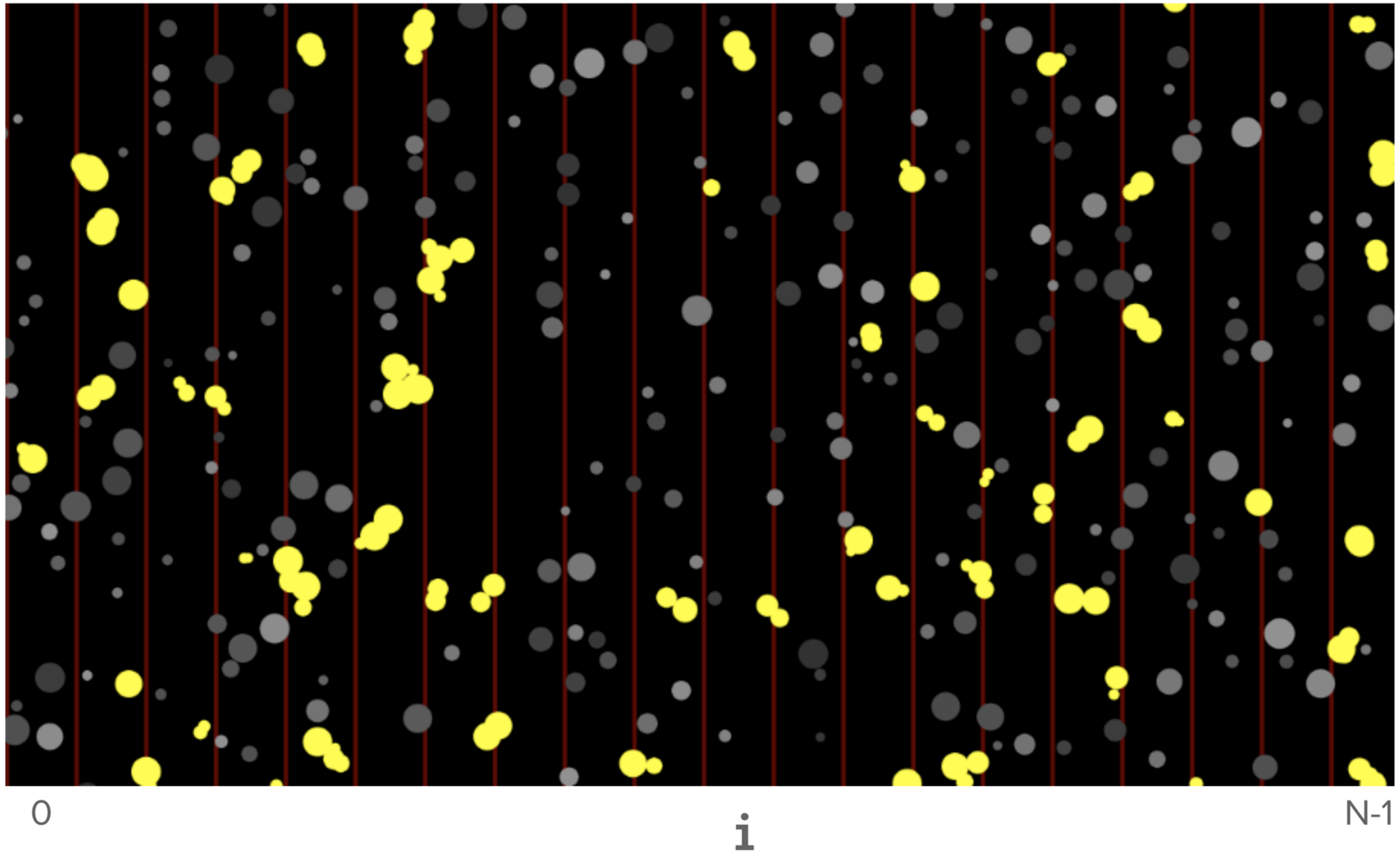
- Bucketing (a.k.a. binning) to the rescue.
  - O(N) binning cost
  - Generalizes beyond 1D
  - Bucket size is an important parameter (why?)

1D Uniform Spatial Subdivision

Construction:

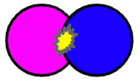
- Divide space into N bins of equal width, h
- Add each object to each bin that its bounding volume overlaps:
  - Use 1D overlap test

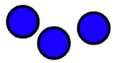
Cell Index, i: Given coordinate x, find containing cell `index(x)` using `Math.floor(x/h)` clamped to [0,N-1].



1D Uniform Spatial Subdivision

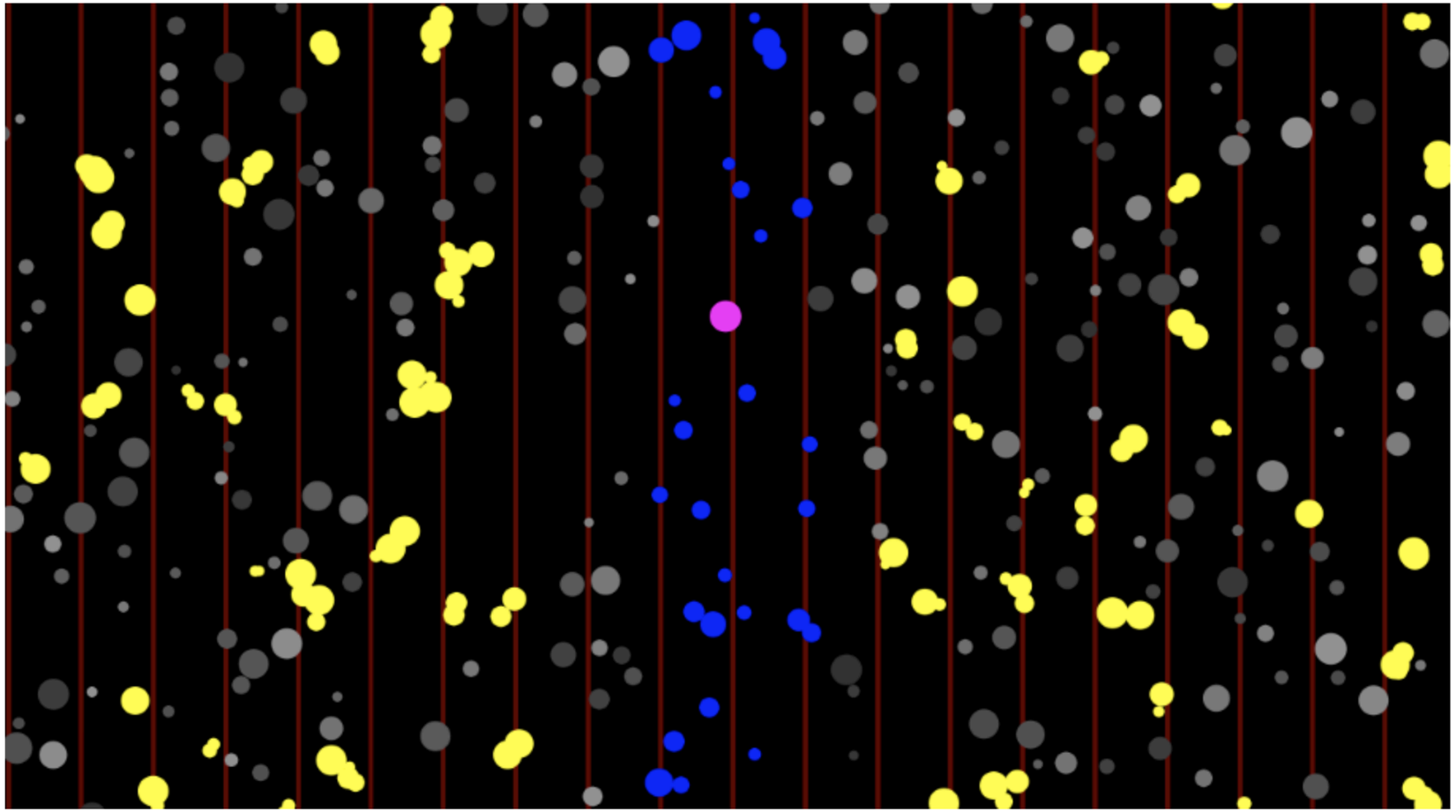
Overlap Testing:

- 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?



2D Uniform Spatial Subdivision

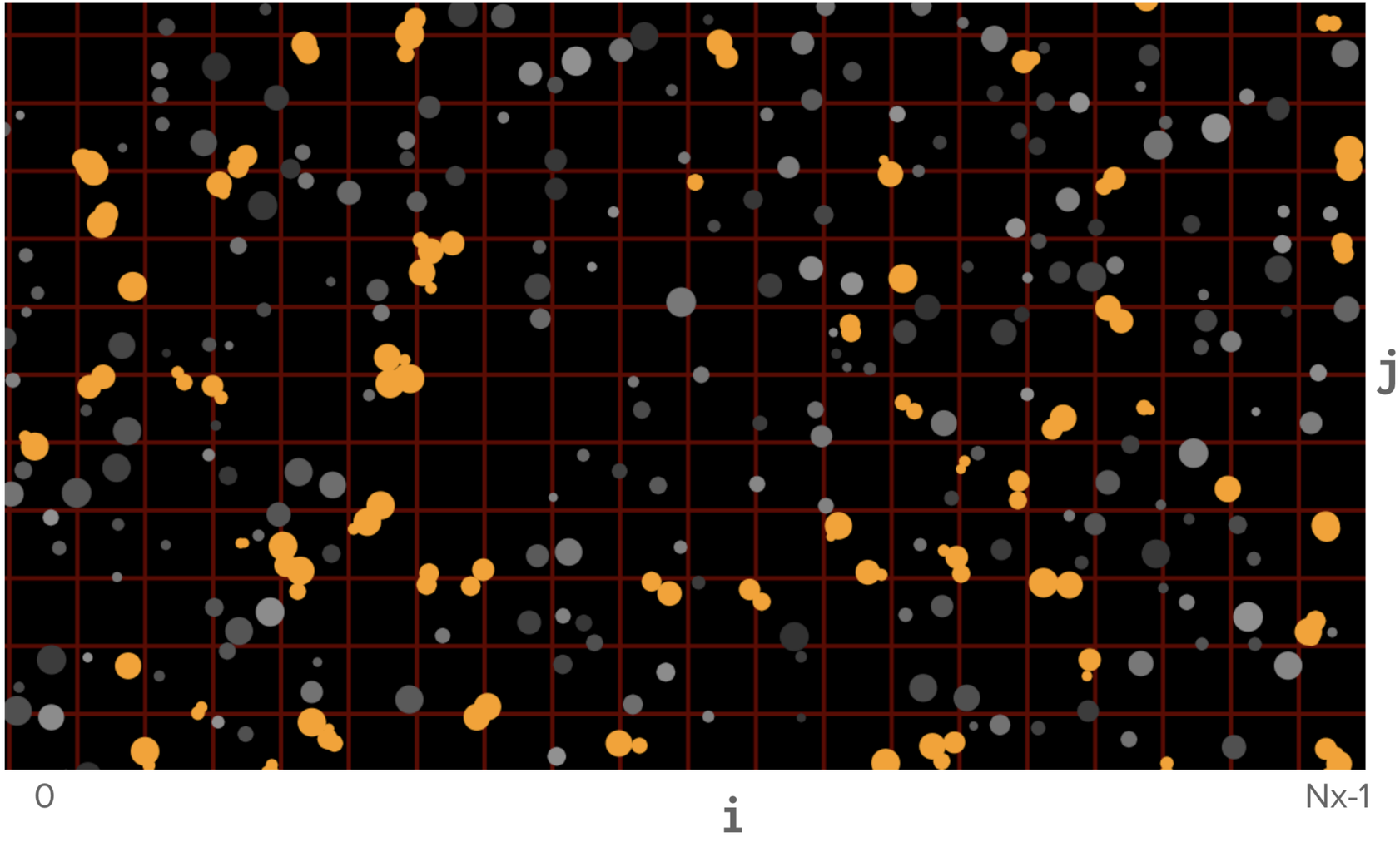
Construction:

- 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,

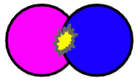
`i = floor(x/h`$_x$`)` clamped to [0,Nx-1],

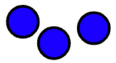
`J = floor(y/h`$_y$`)` clamped to [0,Ny-1].



2D Uniform Spatial Subdivision

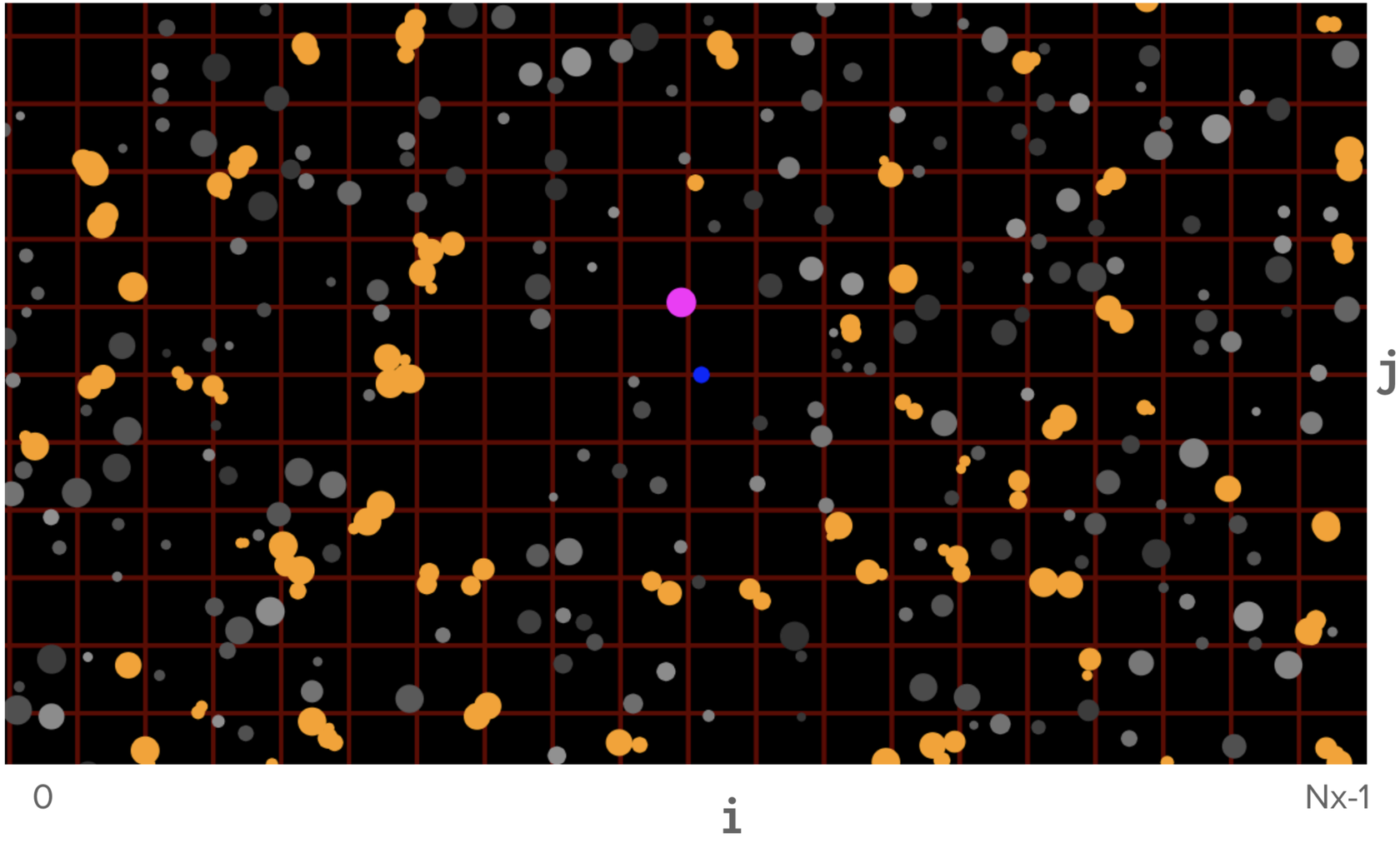
Overlap Testing:

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

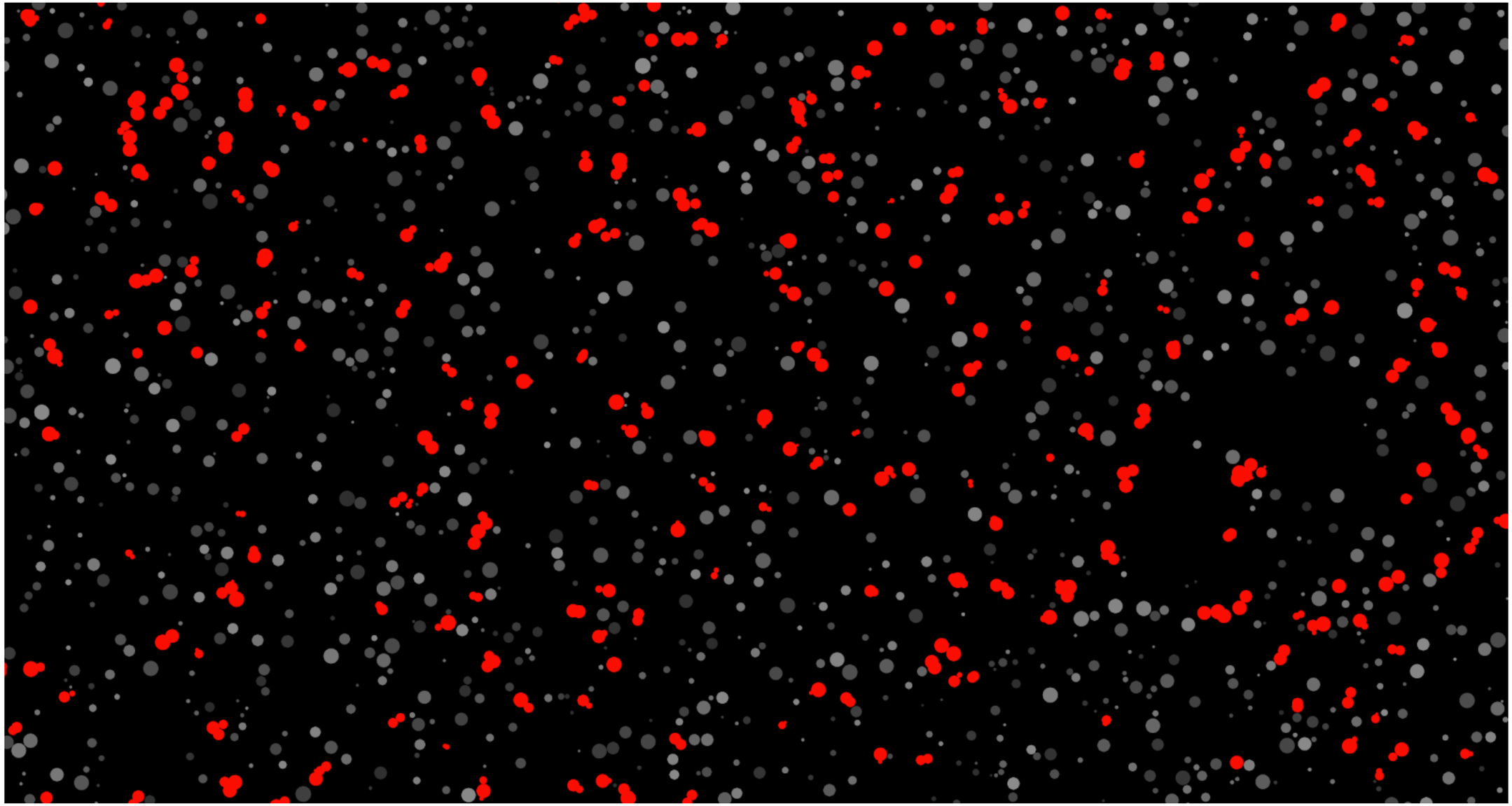


Q: Can duplicate overlaps occur?

Weakness of 2D subdivision?



DEMO: N-Particles Overlap Tests with 1D & 2D Uniform Subdivisions



https://www.openprocessing.org/sketch/973557

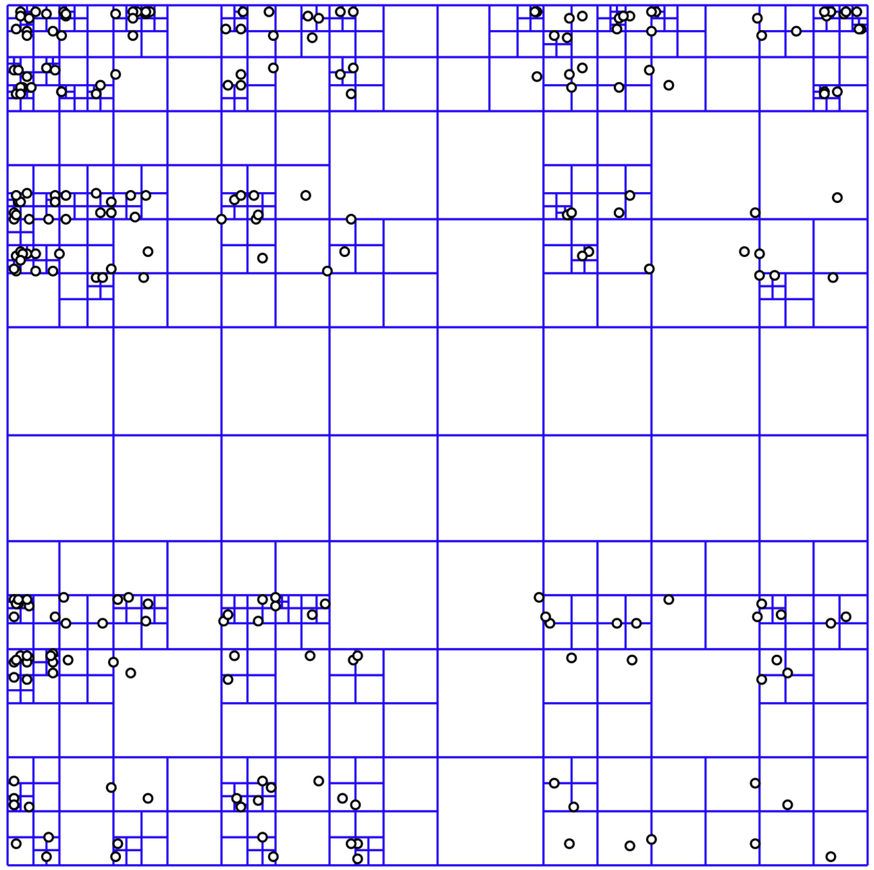
2D Uniform Spatial Subdivision

`GridCheck2D:`

- Simplified implementation. Just drop in your sketch.
- See Sketch: https://www.openprocessing.org/sketch/973557

```
// A simple uniform 2D subdivision for interval-overlap tests on [0,width]x[0,height].
// Given an object and rectangular interval, [xmin,xmax]x[ymin,ymax], the object is stored in cells it overlaps.
// A test rectangle can be used to find objects with overlapping intervals.
// Doug L. James, Sept 2020, http://graphics.stanford.edu/~djames
class GridCheck2D {

  constructor(nXCells, width, nYCells, height) {
    this.nXCells       = nXCells;
    this.nYCells       = nYCells;
    this.nXCellsMinus1 = nXCells-1; // used often
    this.nYCellsMinus1 = nYCells-1; // used often
    this.dx     = width /this.nXCells;
    this.dy     = height/this.nYCells;
    this.invDX  = 1/this.dx; // used often
    this.invDY  = 1/this.dy; // used often
    this.width  = width;
    this.height = height;
```

Adaptive spatial subdivisions



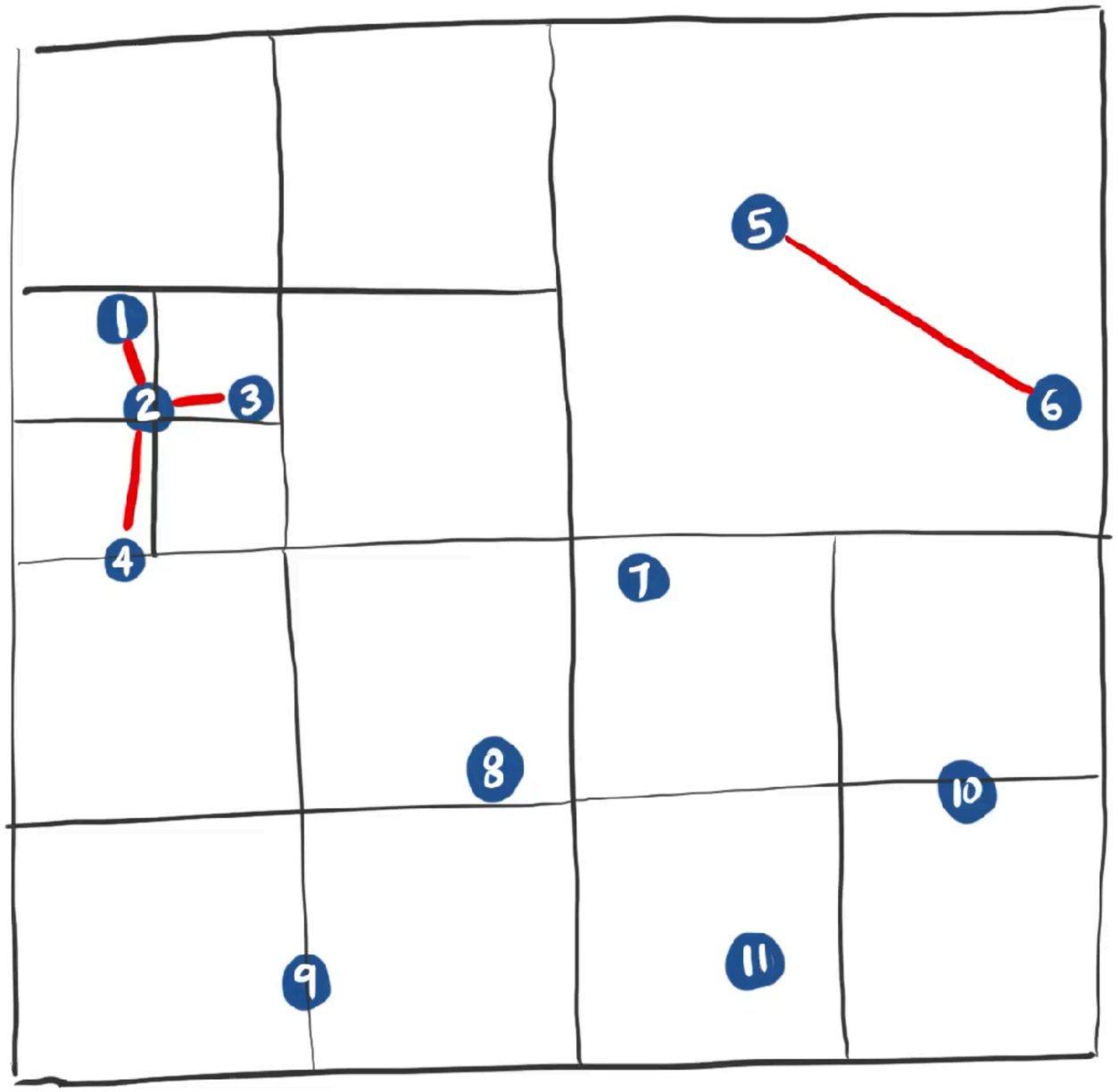
A point-region quadtree with point data. Bucket capacity 1.
https://en.wikipedia.org/wiki/Quadtree

- Adapts bucket sizes to object distribution
- Example: Quadtrees (2D) (octrees in 3D)
  - Construction:
    - Recursively quadrisect cells until
      1. #objects per bucket is $\leq N_{max}$, or
      2. reach maximum tree depth, d_{max}
    - Store objects IDs in list at leaf nodes
  - Query: Traverse leaves to gather set of potentially overlapping object pairs.
- Other spatially adaptive data structures:
  - kd-trees: Binary tree. Popular in data science.
  - Hierarchical grids: Good for very different size objects
  - Hybrid: E.g., octree with uniform subdivision at leaf nodes

EX: Quadtree (N_{max}=2)

PAIRS

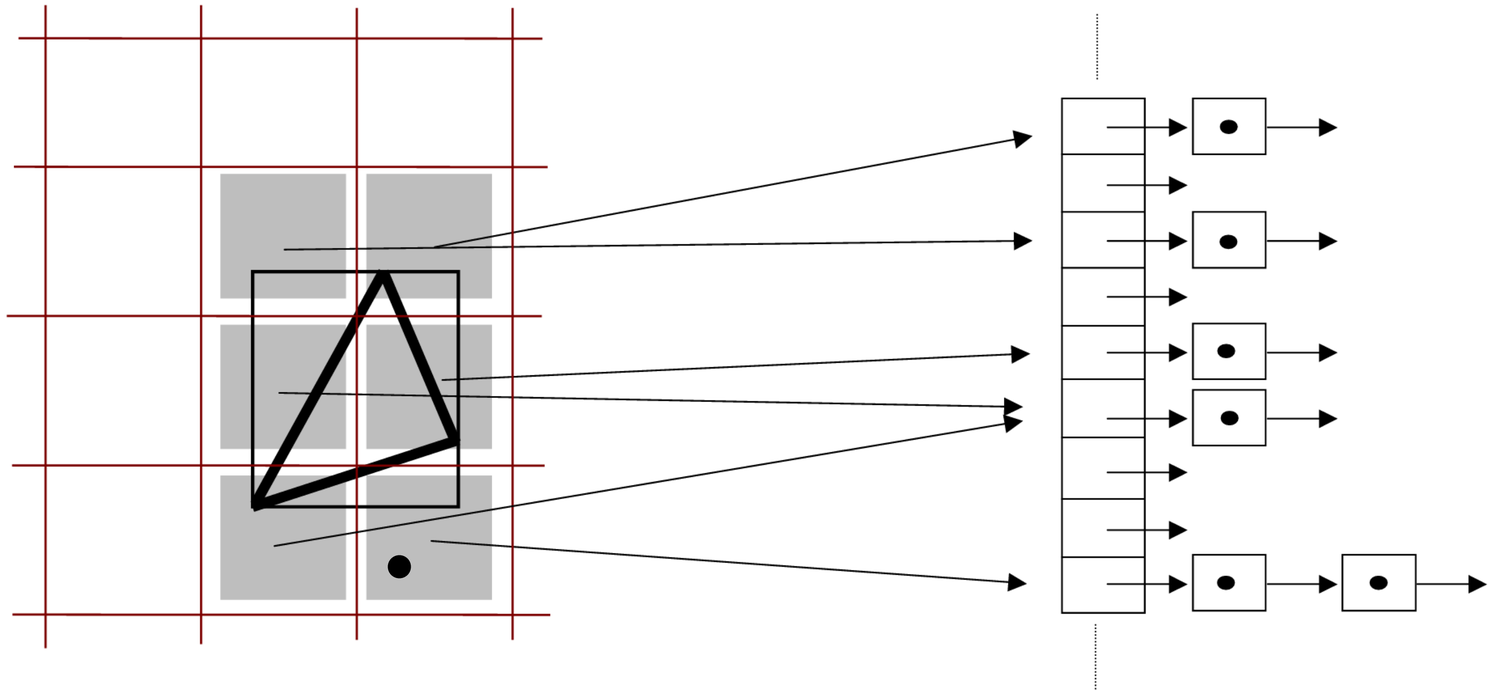
(1,2)
(2,3)
(2,4)
(5,6)



Recall: Uniform subdivisions

- Popular
- PROS:
  - Simple
  - Often fast
- CONS:
  - Fixed domain size 🙄
  - Memory intensive due to dense grid 😩
  - Yet potentially many empty cells 😭
  - Single scale

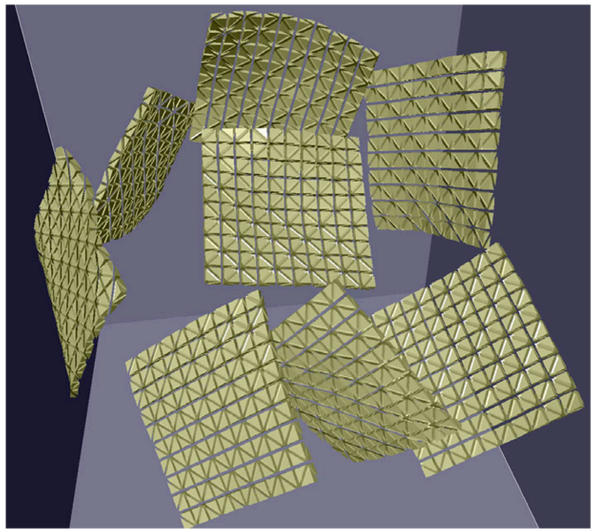
Spatial Hashing



Hash values are computed for all grid cells covered by the AABB of a tetrahedron [Teschner et al. 2003]

- Uniform grid of unbounded size.
- Uses a hashtable instead of dense array.
- Hash function maps cell indices to a list of cell entries.

Spatial Hashing: Choice of hash function



- Maps cell indices, e.g., (i, j, k) in 3D, to array of size n.
  - Let hash function, $\text{hash}(i, j, k; n)$, map to integer on [0,n).
- Example from [Teschner et al. 2003]

$$\text{hash}(i, j, k; n) = \left(i\, p_1 \text{ XOR } j\, p_2 \text{ XOR } k\, p_3\right) \mod n$$

where p_1, p_2, p_3 are large prime numbers,

e.g., $p_1 = 73856093$, $p_2 = 19349663$, $p_3 = 83492791$.

- Hashcode collisions:
  - Different cells can map to the same index. Hopefully rare.
- Collision checks on each hashtable list:
  - Verify same cell indices. Do overlap tests.

Wait! What about negative i, j, k indices???

- Recall we want

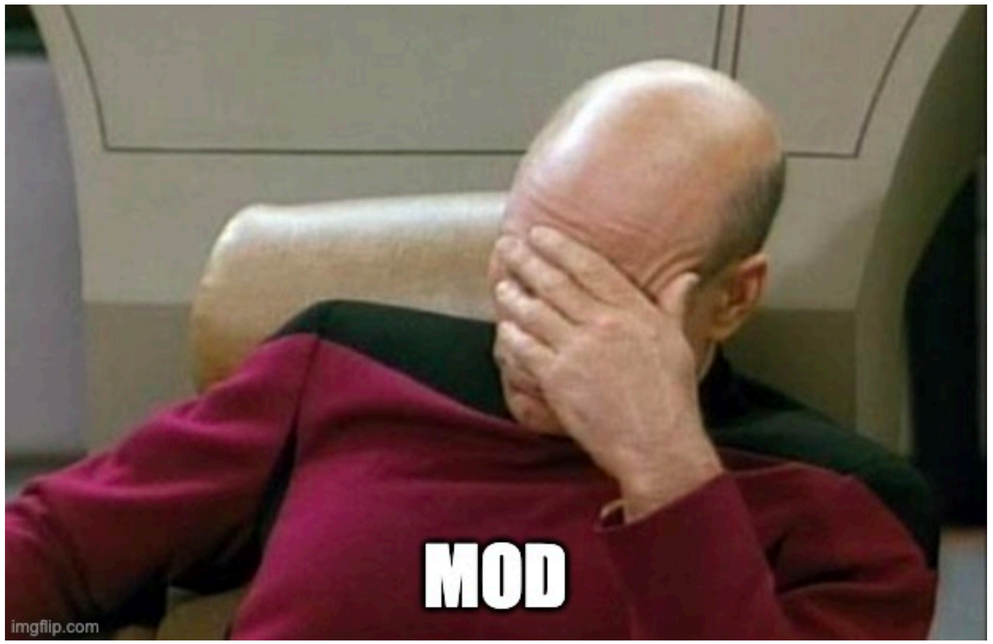
$$\text{hash}(i, j, k; n) = \left(i\,p_1 \text{ XOR } j\,p_2 \text{ XOR } k\,p_3\right) \bmod n$$

  to map to [0,n).

- Things to consider:
  - XOR with negative numbers
  - mod with negative numbers
    - Recall we want a nonnegative result

Which mod function(!?!)

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



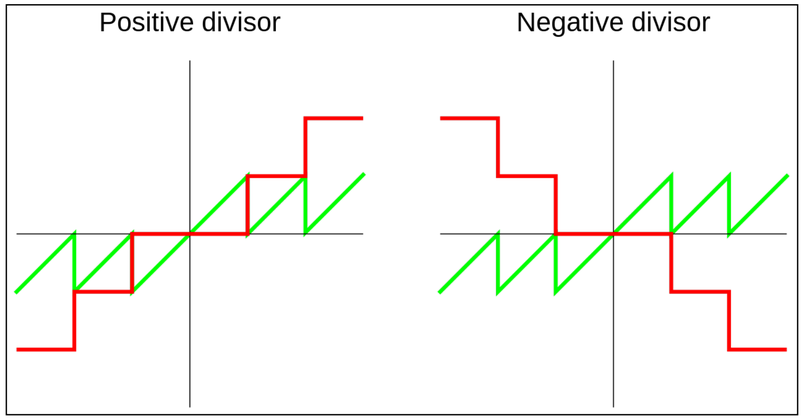
Which mod function(!?!)

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

- Mathematically, *a modulo n* is interpreted as the smallest non-negative integer remainder, r
- But, in nearly all computer systems, the quotient q and remainder r of a divided by n satisfy

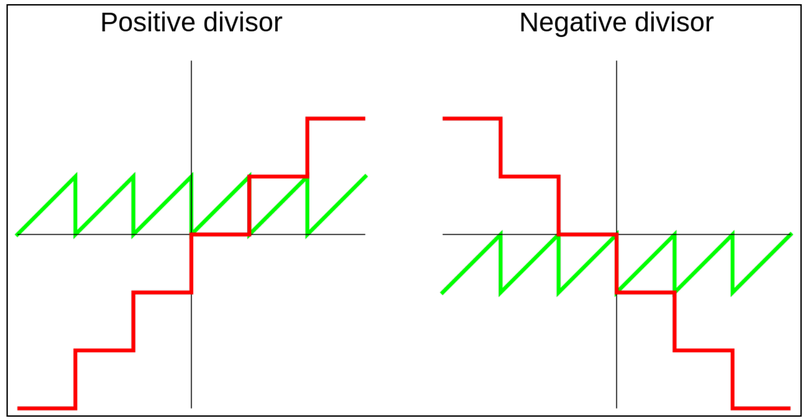
$$a = nq + r, \quad \text{where} \quad q \in \mathbb{Z} \quad \text{and} \quad |r| < n$$

- Also, there is no agreement by programming languages on which r definition to use.
- Popular choices: truncated, floored, Euclidean, rounded, etc.



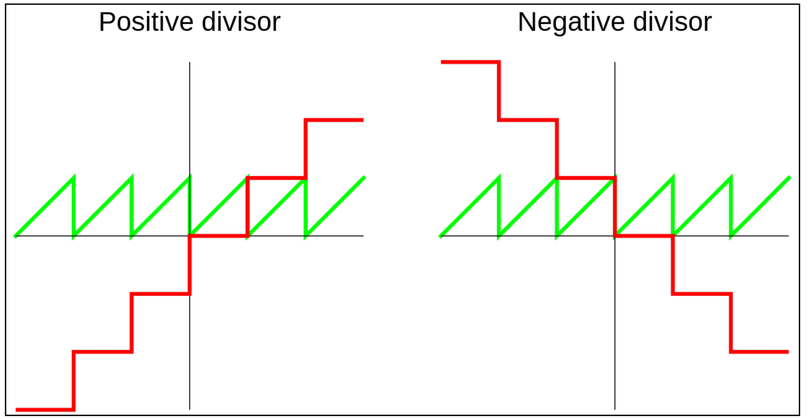
truncated division

$$r = a - n \operatorname{trunc}\left(\frac{a}{n}\right)$$



floored division

$$r = a - n \left\lfloor \frac{a}{n} \right\rfloor$$

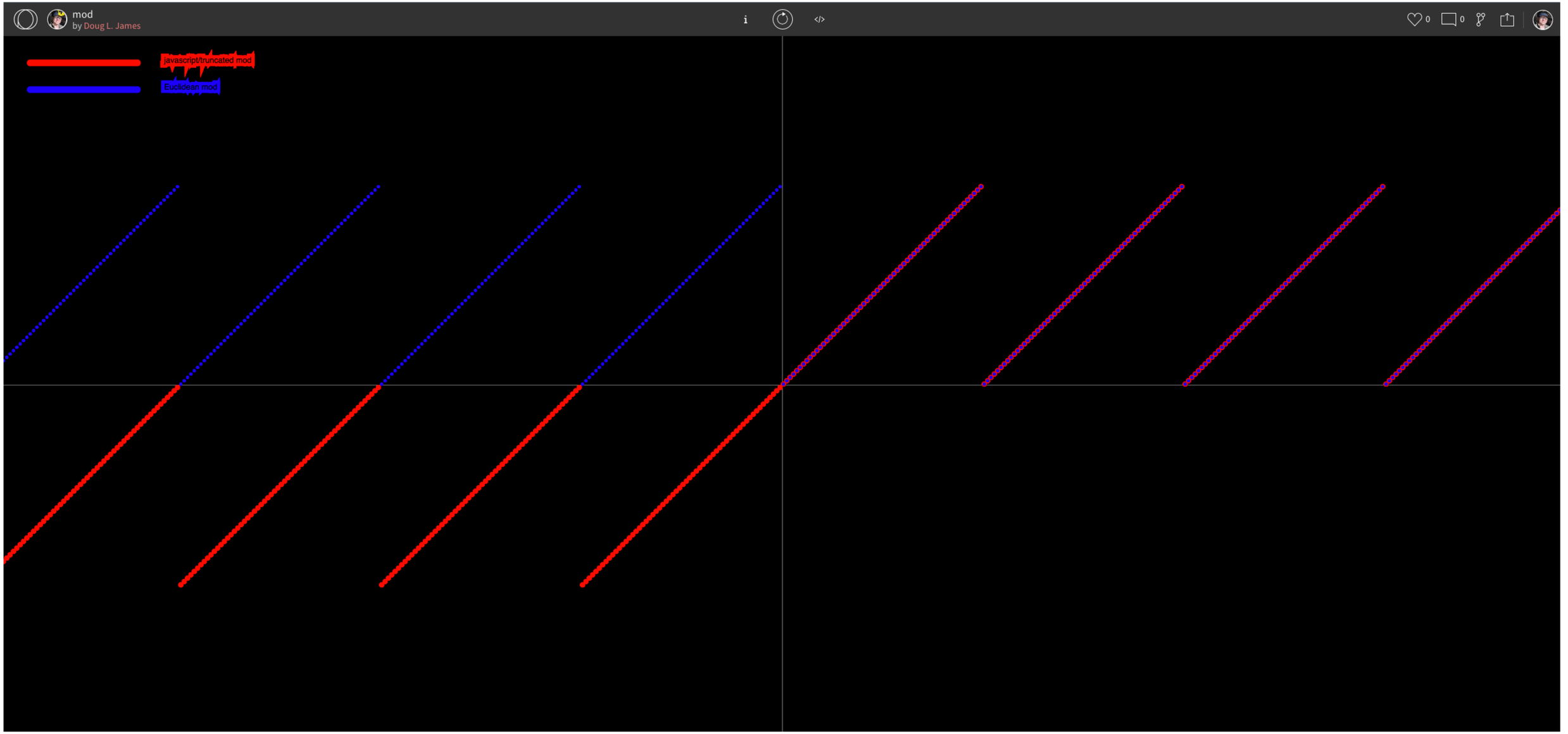


❤️ **Euclidean division**

$$r = a - |n| \left\lfloor \frac{a}{|n|} \right\rfloor$$

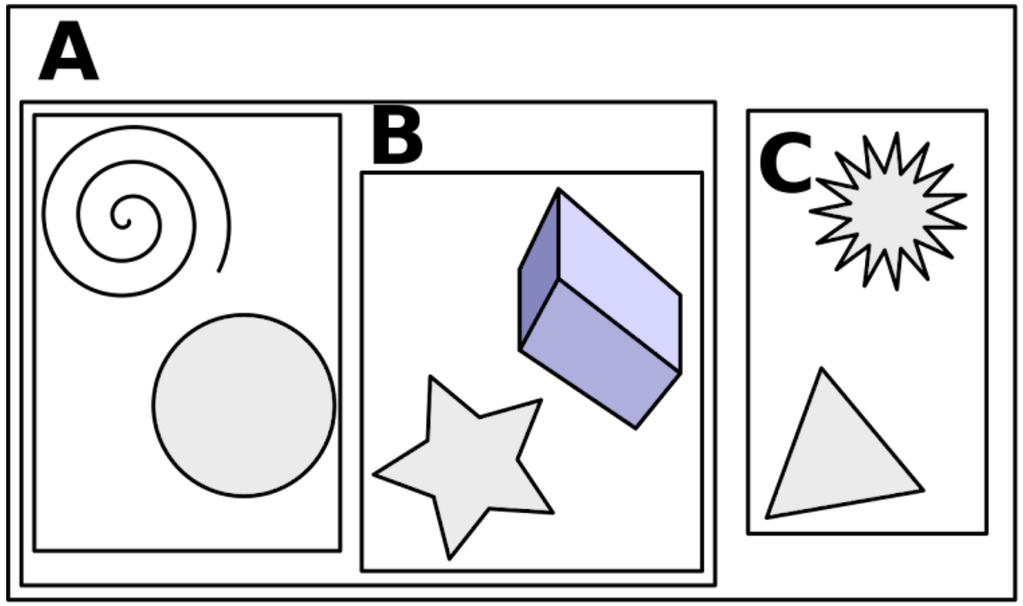
Which mod function does JavaScript use??

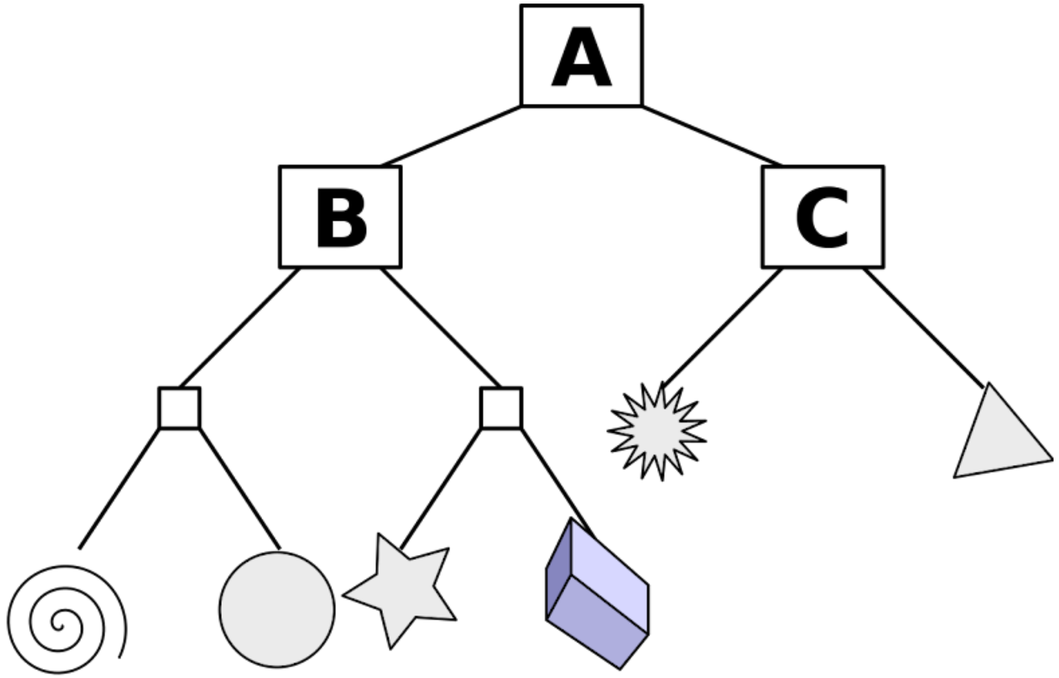
https://openprocessing.org/sketch/1692537



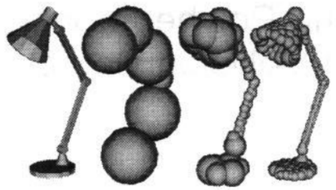
Bounding Volume Hierarchies

Bounding Volume Hierarchies (BVHs)



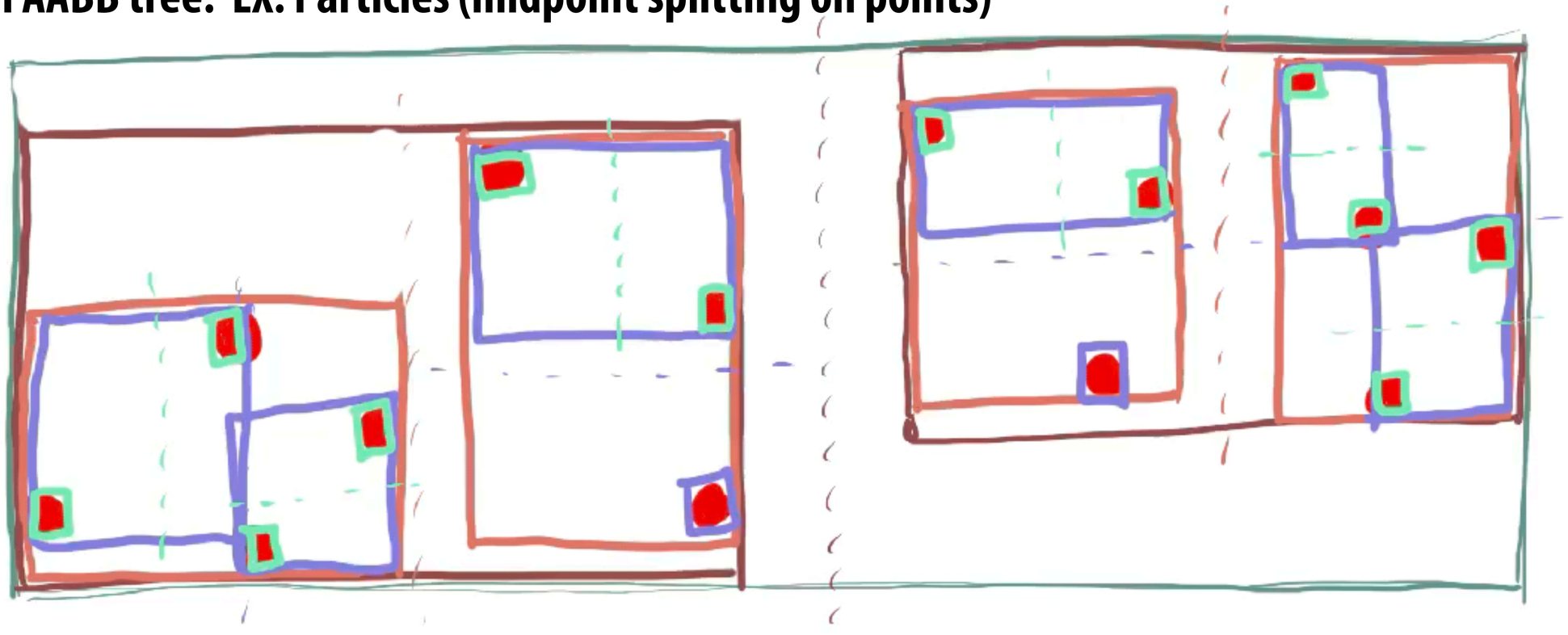


- **Common BVHs in graphics:**
  - **AABB Trees [van den Bergen 1998]: good for deformable models**
  - **OBB Trees [Gottschalk et al. 1996]: tighter fitting**
  - **Sphere Trees [Hubbard 1996]**



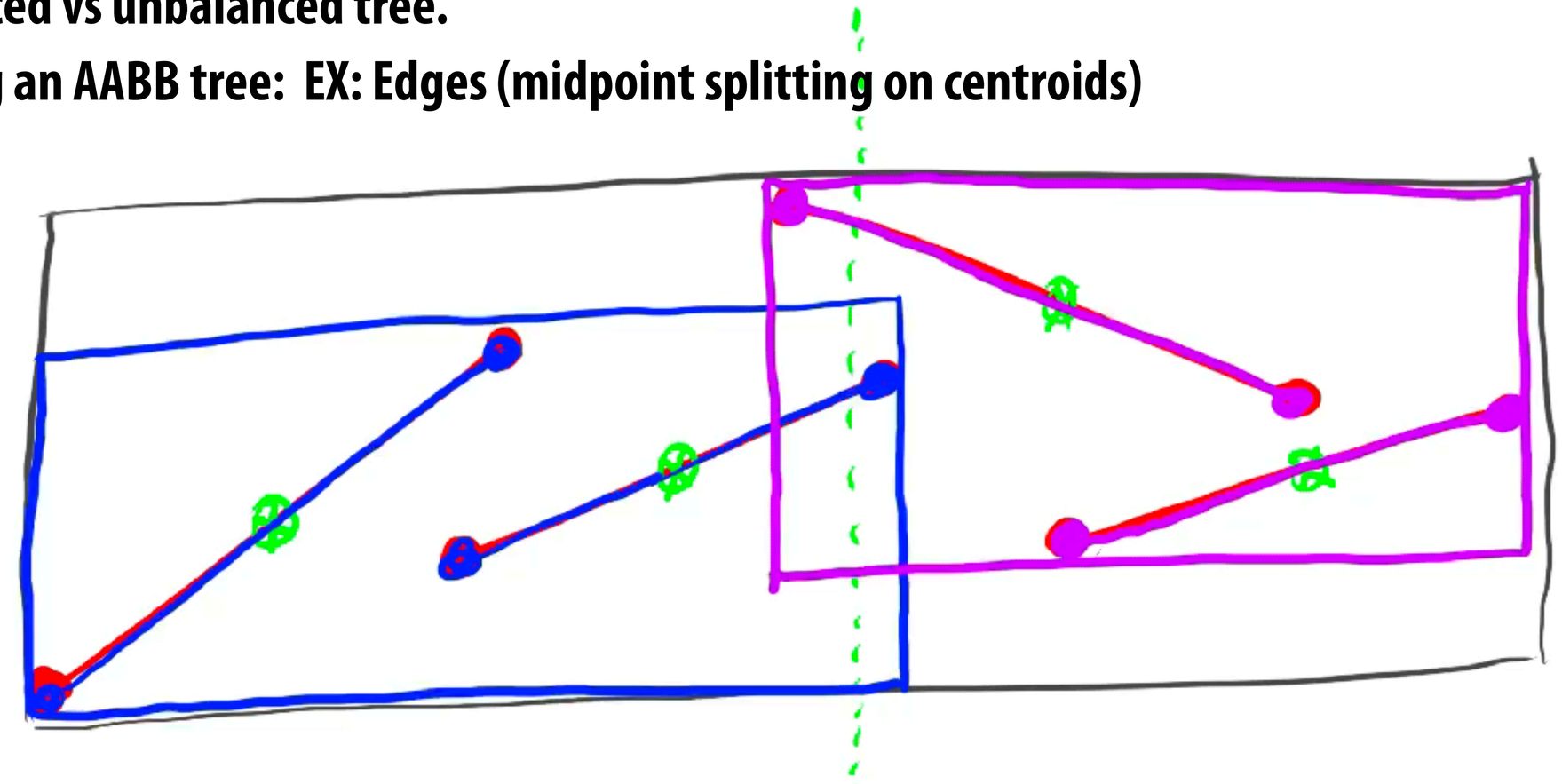
Bounding Volume Hierarchies (BVHs)

- Object-based adaptive spatial data structure
- Construction (binary tree):
  - Top down: Recursively partition then bound. O(N lg N) work.
    - Binary tree: Split along largest dimension or diameter. Choice of "middle" (mean vs median). Balanced vs unbalanced tree.
    - Fitting an AABB tree: EX: Particles (midpoint splitting on points)



Bounding Volume Hierarchies (BVHs)

- Object-based adaptive spatial data structure
- Construction (binary tree):
  - Top down: Recursively partition then bound. O(N lg N) work.
    - Binary tree: Split along largest dimension or diameter. Choice of "middle" (mean vs median). Balanced vs unbalanced tree.
    - Fitting an AABB tree: EX: Edges (midpoint splitting on centroids)



BVH Fit: Wrapped vs Layered Hierarchies

- Wrapped BVH nodes tightly fit the underlying geometry
- Layered BVH nodes enclose their child nodes, or geometry when leaf nodes.
- AABB trees: No difference.
- Sphere trees:
  - Clear difference in tightness ⟶
  - Fitting cost?

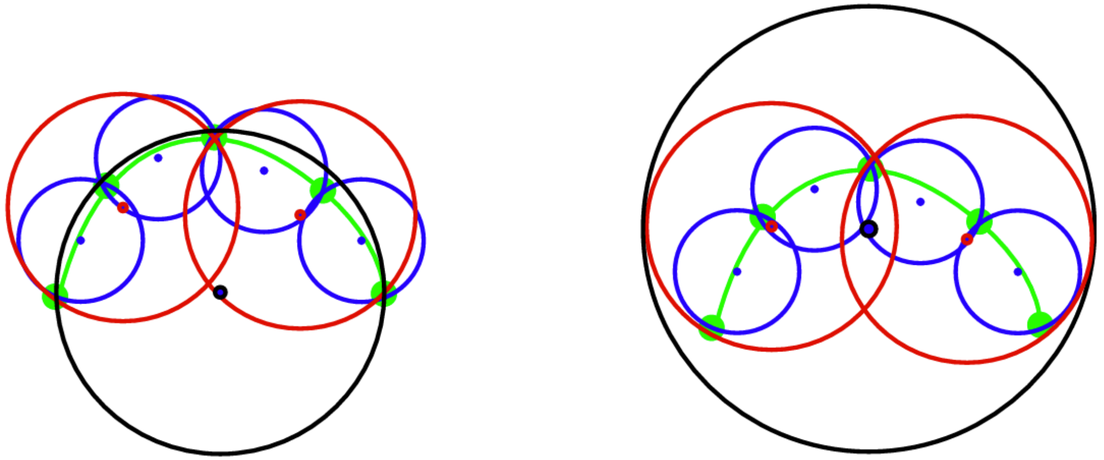


Figure 3: *The wrapped hierarchy* (left) has smaller spheres than the layered hierarchy (right). The base geometry is shown in green, with five vertices. Notice that in a wrapped hierarchy the bounding sphere of a node at one level need not contain the spheres of its descendents and so can be significantly smaller. However, since each sphere contains all the points in the base geometry, it is sufficient for collision detection.

From "Bounded Deformation Trees" [James and Pai 2004]

BVH Fit: Wrapped vs Layered Binary Sphere Trees

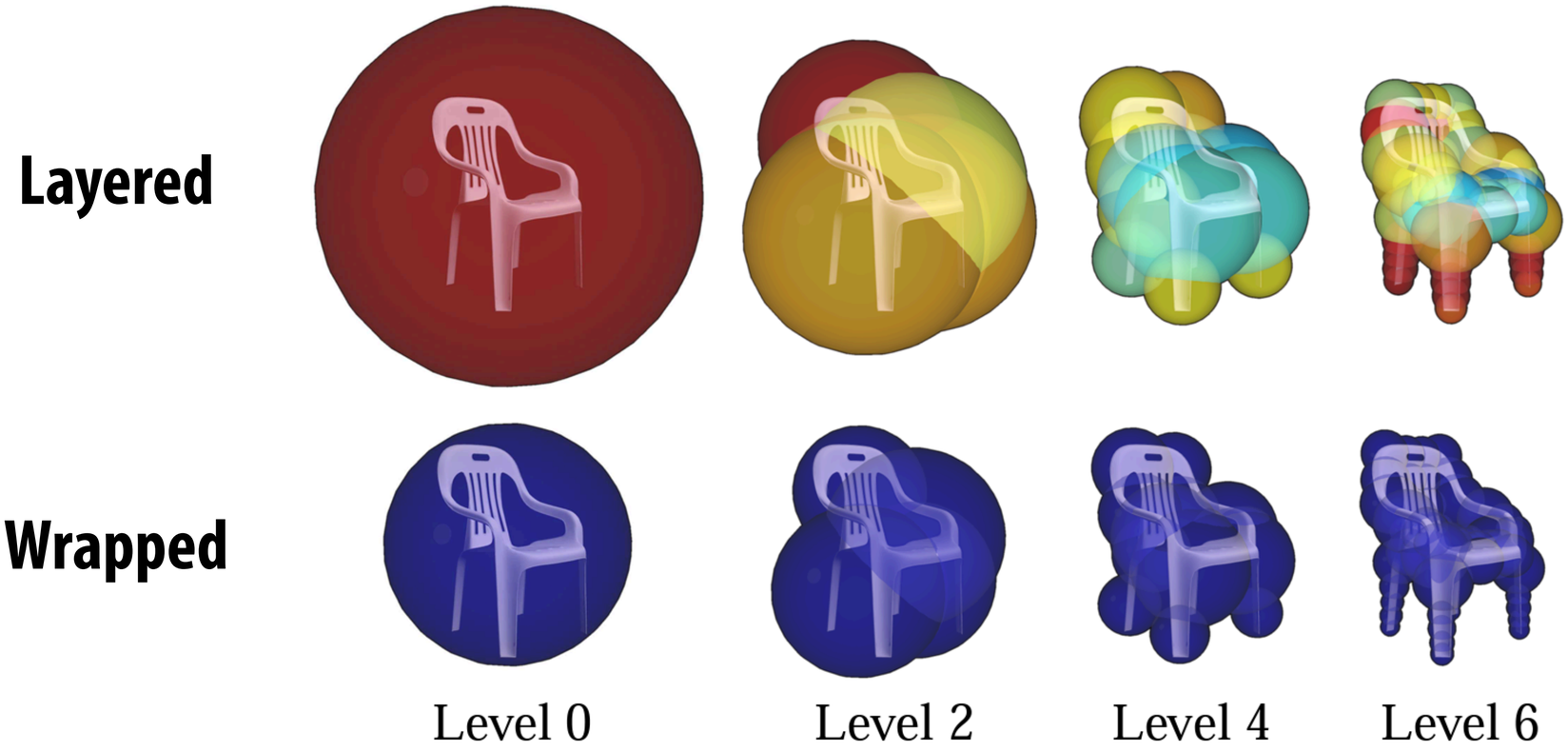
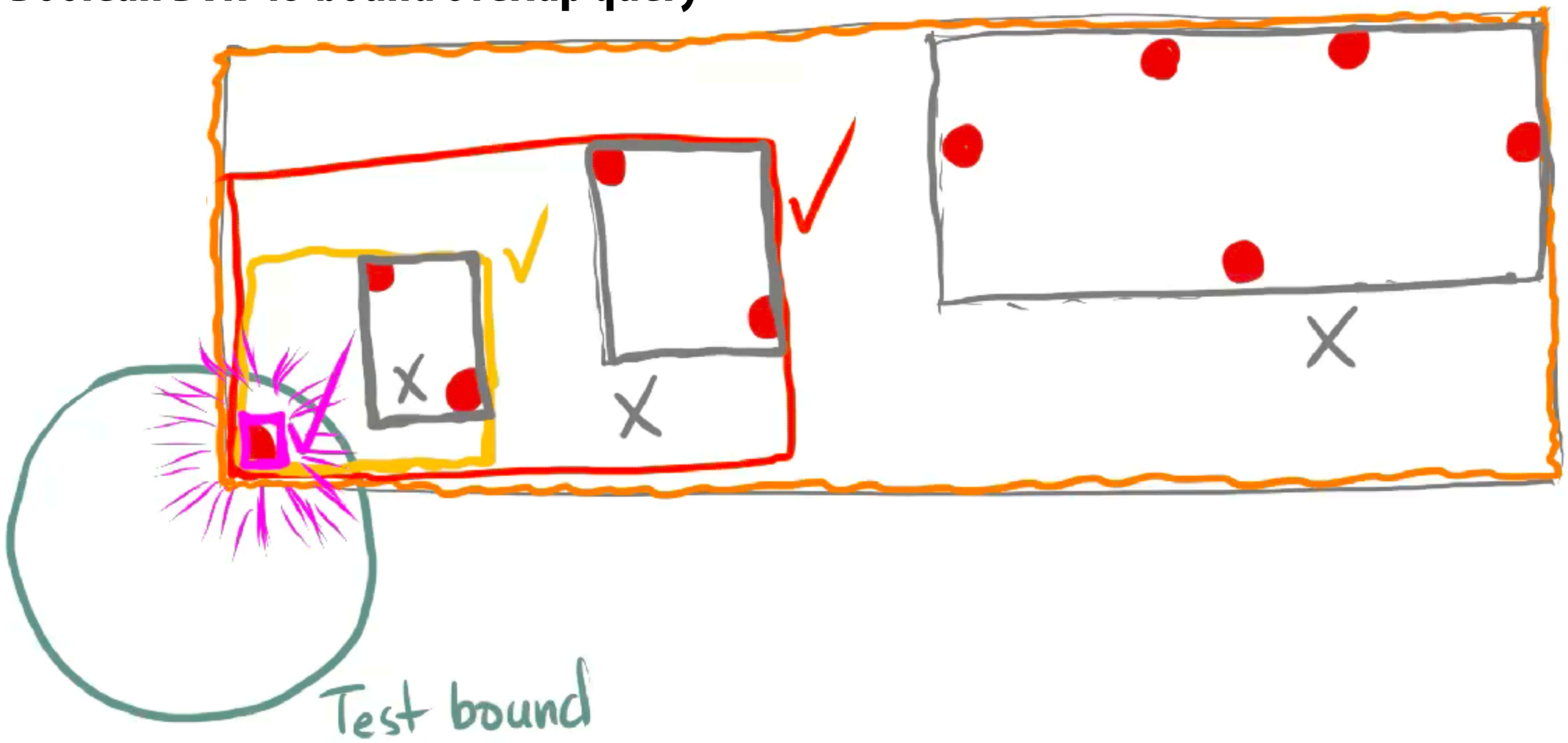


Figure 4: *Comparison of undeformed layered hierarchy (top) versus the wrapped BD-Tree (bottom).* Throughout the paper, dark red denotes at least 3 times greater sphere volume (or $\geq 1.44\times$ radius) *than the undeformed wrapped BD-Tree sphere*, and dark blue implies a negligible size increase.

From "Bounded Deformation Trees" [James and Pai 2004]

BVH Overlap Queries

- Object-based adaptive spatial data structure
- Query (binary tree):
  - Recursive on intersecting bounds until leaf overlap(s) found.
  - EX: Boolean BVH-vs-bound overlap query



BVH Overlap Queries

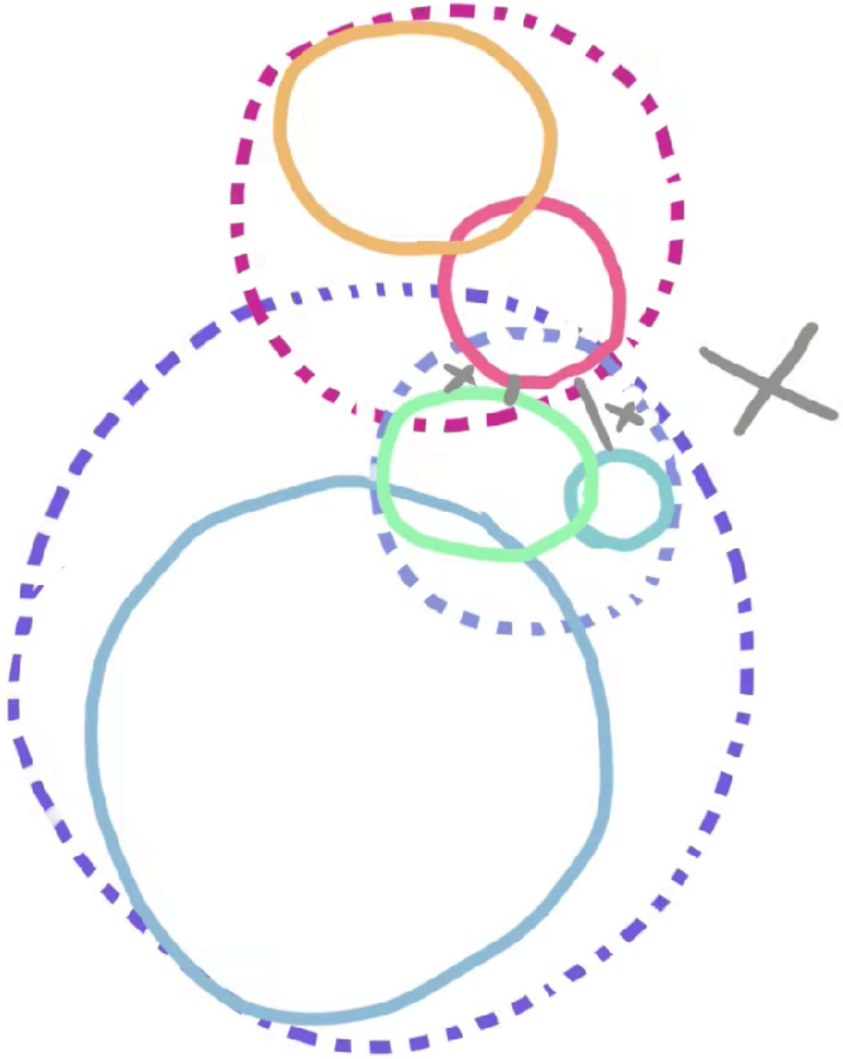
- Object-based adaptive spatial data structure
- Query (binary tree):
  - Recursive on intersecting bounds until leaf overlap(s) found.
  - EX: Boolean BVH-vs-bound overlap query

```
// Recursive binary-BVH-tree overlap query against a bound.
function overlaps(node, bound) {
  if(node==null) return false;
  if(node.intersects(bound)) {
    if(node.isLeaf())
      return true; // or test leaf geometry
    else
      return overlaps(node.left, bound) || overlaps(node.right, bound);
  }
  else
    return false;
}
```

BVH Overlap Queries

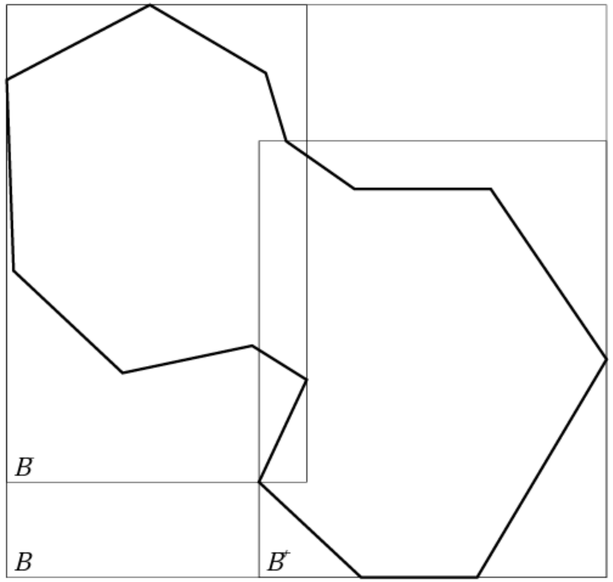
- **Query (binary tree): Boolean BVH-vs-BVH overlap query**

```
// Recursive binary-BVH vs BVH overlap query.
function overlaps(a, b) {
  if(a==null || b==null) return false;
  if(a.intersects(b)) {
    if(a.isLeaf() && b.isLeaf()) // we made it!!
      return true; // or test leaf geometry
    else if (a.isLeaf()) // split b
      return overlaps(a, b.left) || overlaps(a, b.right);
    else if (b.isLeaf()) // split a
      return overlaps(b, a.left) || overlaps(b, a.right);
    else { // split largest node first:
      let big = (a.size > b.size) ? a : b;
      let lil = (a.size > b.size) ? b : a;
      return overlaps(big.left, lil) || overlaps(big.right, lil);
    }
  }
  else // bounds don't intersect so no overlap
    return false;
}
```

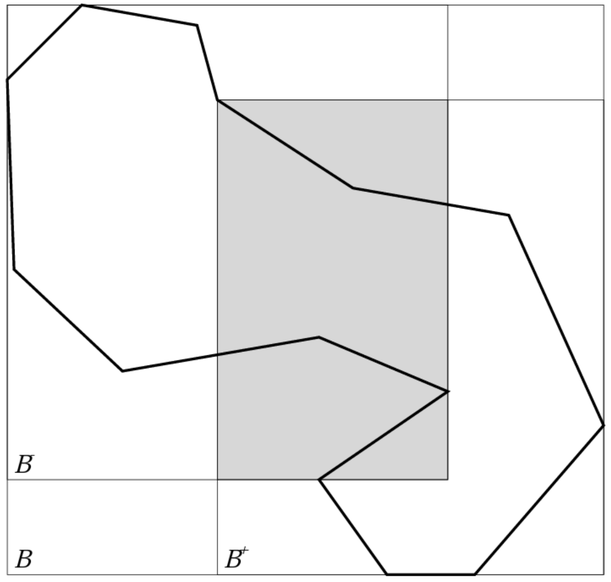


Updating BVHs after deformation

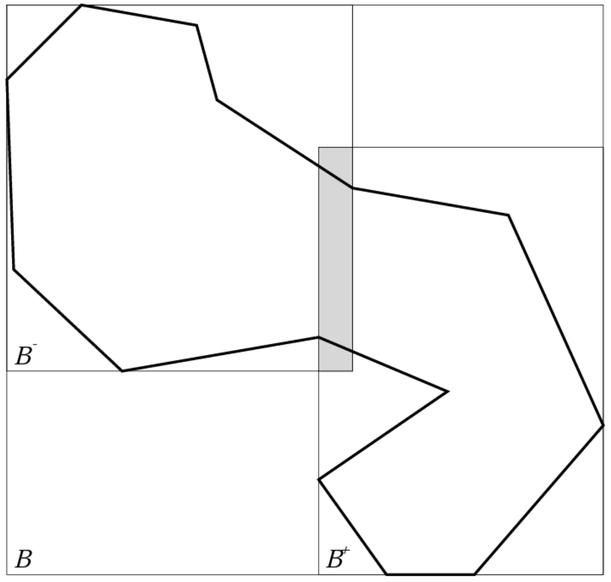
- Deformable objects need their BVHs updated each timestep
  - Major overhead, especially when there are no collisions.
- AABB Trees are a popular choice [van den Bergen 1997]
  - Cheap bounds to fit
  - Two approaches:
    - Rebuild the tree (costly, O(N lg N)). For large deformations.
    - Refit all bounds (cheaper, O(N)). For small deformations.



Undeformed



(a) Refi tted



(b) Rebuilt

Advanced

Output-sensitive updating of BVHs after deformation

Requires reduced-order deformation models

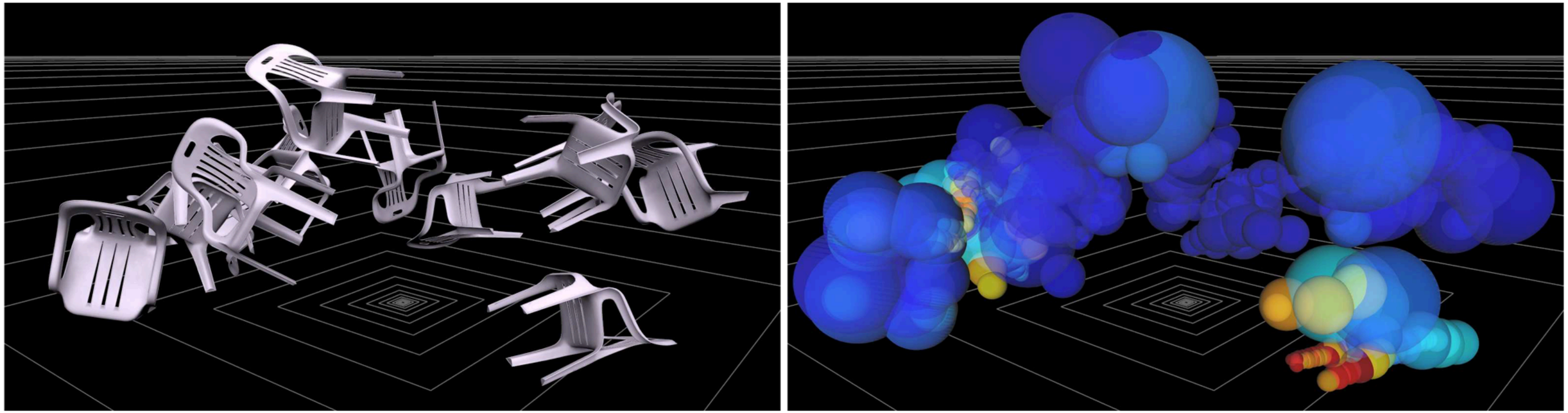
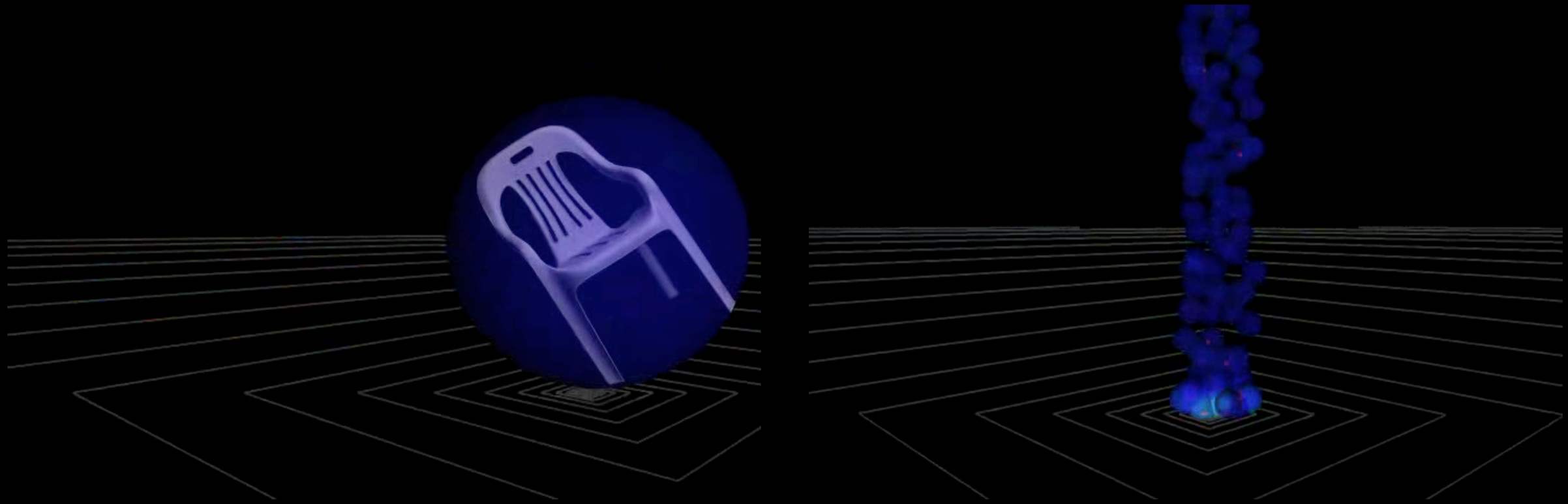


Figure 7: *Output-sensitive deformable collision processing:* (Left) simulation of 12 chairs crashing to the ground; (Right) updated spheres illustrating the extent of sphere tree traversal. Only a small fraction of BD-Tree spheres are updated at any time step. The tree traversal depth can be bounded to lower collision processing costs (shown for maximum depth of 8).

From "Bounded Deformation Trees" [James and Pai 2004]

Output-sensitive updating of BVHs after deformation

Requires reduced-order deformation models



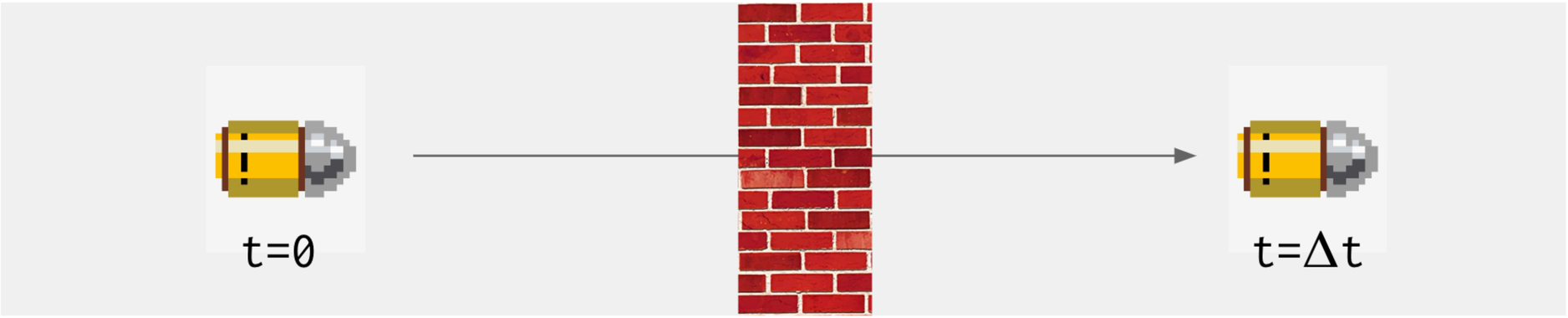
From "Bounded Deformation Trees" [James and Pai 2004]

Continuous Collision Detection

Motivation: Discrete vs continuous collision detection

Problem:

Missed collisions

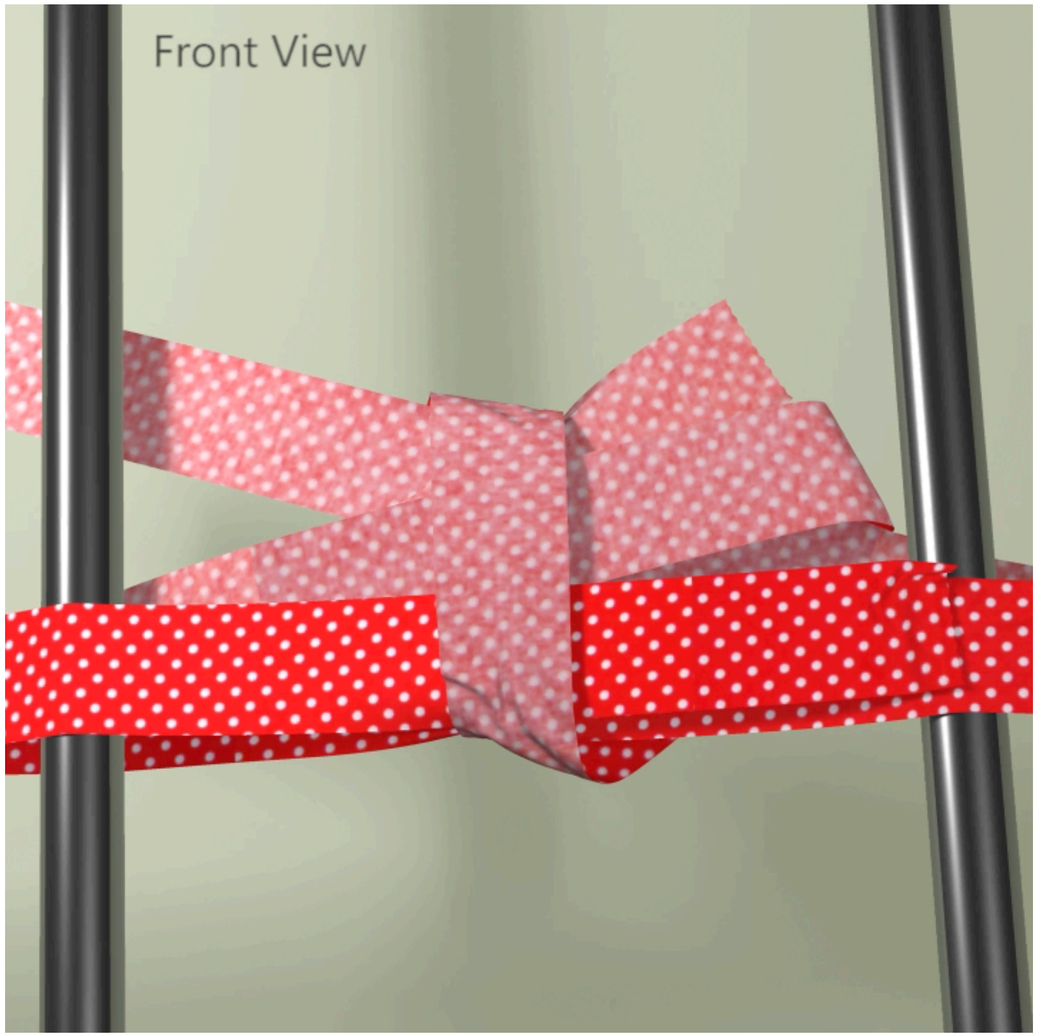


Benefits of continuous time collision checks

- Avoids missed collisions
  - Common in game physics libraries, e.g., Bullet
- Useful for accurate time-of-collision (ToC) estimates
  - Ensures proper ordering of collision events
- Important for simulation of cloth and thin objects
  - Less common for volumetric simulations

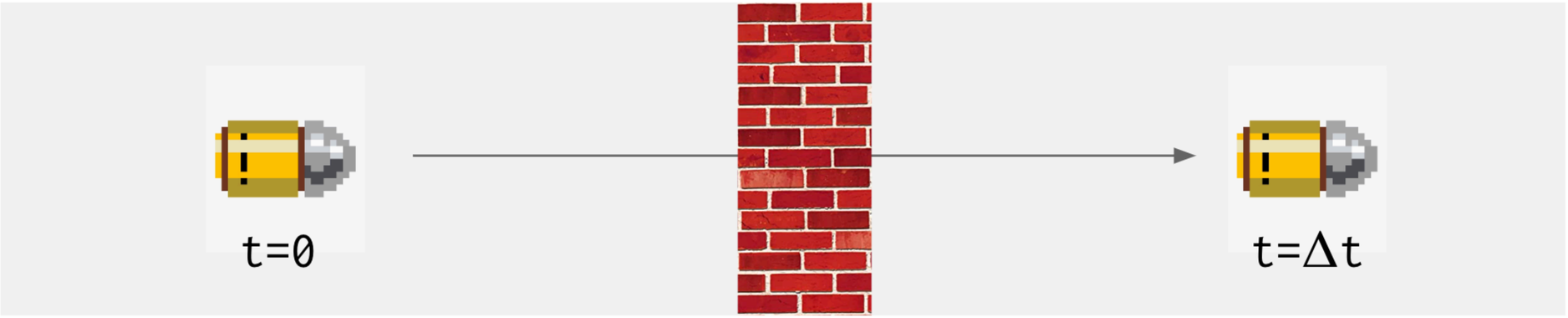
Example: CCD critical for thin structures

[Kaufman et al. 2014]
http://www.cs.columbia.edu/cg/adonis/



[Wang 2014]
https://wanghmin.github.io/

CCD tests are fancy root-finding problems



- Given a scalar function $f(t)$ for some time-like parameter, t.
- Find the *first* time $t^* > 0$ for which $f(t^*) = 0$.
  - Here t^* is the *first* time of collision (ToC)
- Can assume that $f(0) \neq 0$, i.e., collision free at start.
- May be other contact conditions to check, e.g., intersection point is on the line segment.

CCD Roadmap

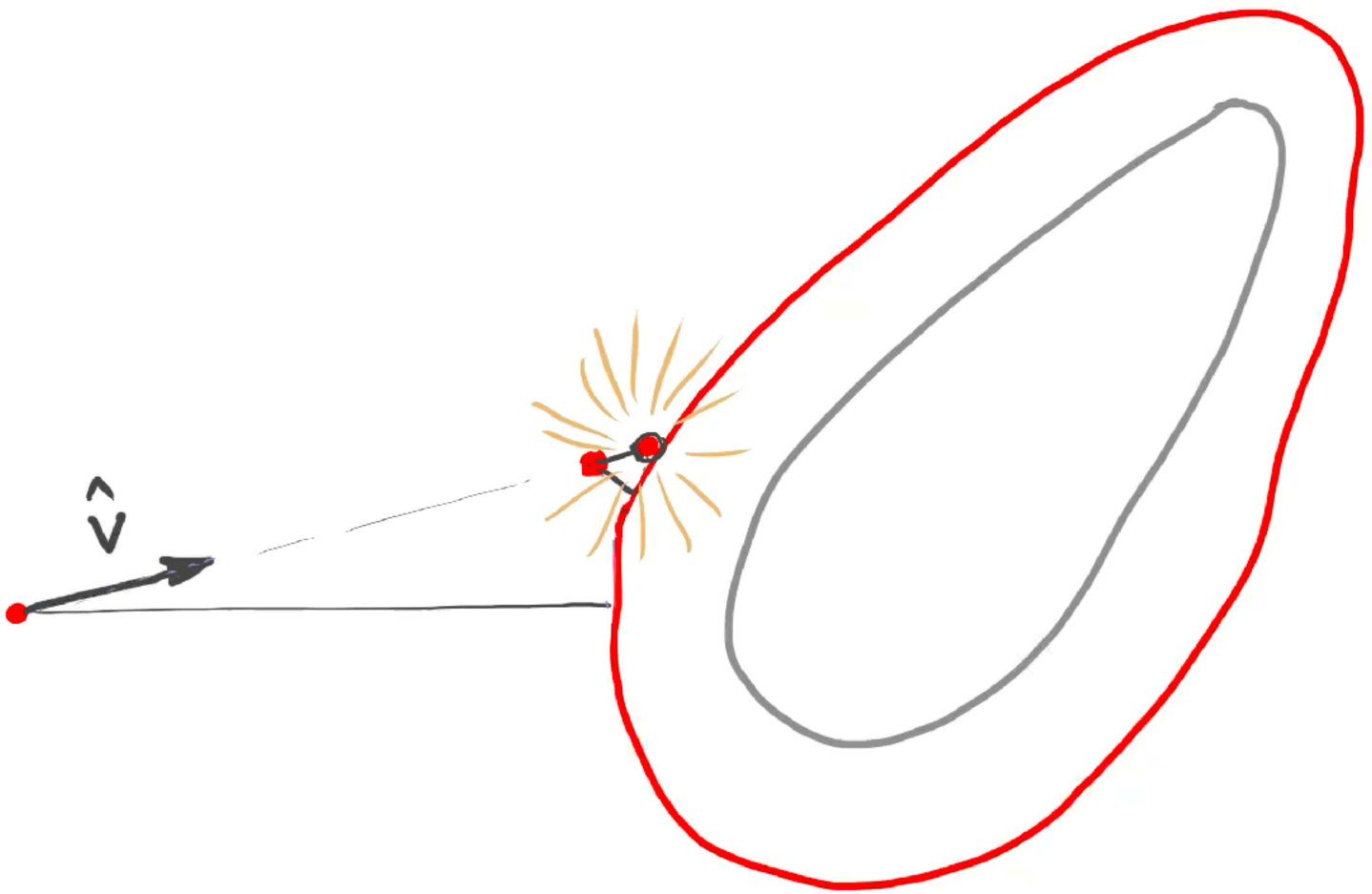
- Pinball: Ray-marching ball-SDF collisions.
  - ball-SDF
  - We need this for P1
- Some common CCD tests
- Robust root-finding methods

Ray-marching ball-SDF collisions

Pinball: Ray-marching ball-SDF collisions

- **Problem: Find all fast-moving ball collisions with a rigid SDF scene during a timestep.**
- **Given**
  - **initial ball position, $\mathbf{p}_0$, velocity, $\mathbf{v}$, radius, r, and timestep size, Δt.**
  - **radius-corrected signed-distance field, $\mathrm{sdf}(\mathbf{p}) = \mathrm{sdf}_{scene}(\mathbf{p}) - r$**
  - **assume linear trajectory, $\mathbf{p}_0 + \mathbf{v}\,t,\ \ t \in [0, \Delta t]$.**
- **Collision distance value along trajectory is $\mathrm{sdf}(\mathbf{p} + \mathbf{v}\,t),\ \ t \in [0, \Delta t]$.**
- **Depth: In terms of depth travelled along the ray, $\delta \in [0, \|\mathbf{v}\|\ \Delta t]$, the collision distance is $\mathrm{sdf}(\mathbf{p} + \hat{\mathbf{v}}\,\delta),\ \delta \in [0, \|\mathbf{v}\|\ \Delta t]$.**

Pinball: Ray-marching ball-SDF collisions



- **Ray marching algorithm:**

Init:

$$\delta = 0$$

$$\delta_{max} = \|\mathbf{v}\| \, \Delta t$$

while $(\delta < \delta_{max})$

$$\mathbf{p} = \mathbf{p}_0 + \delta \, \hat{\mathbf{v}}$$

$$d = \mathrm{sdf}(\mathbf{p})$$

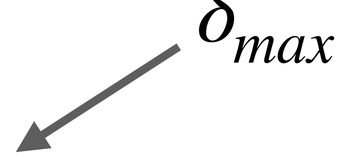
$$\delta \mathrel{+}= d$$

if $(d < \varepsilon)$ **break;**

return δ

Pinball: Ray-marching ball-SDF collisions

- Ray marching pseudocode with limit on max #steps:




```
function rayMarchTimestep(p0, vHat, depthMax) {

  let depth = 0;// init depth along ray

  for (let i=0; i<MAX_MARCHING_STEPS; i++) {
    let   p = p0 + depth*vHat; // ball position
    float d = sdf(p); // max safe dist ball can move
    depth += d; // walk along ray “d”
    if (d < PRECISION || depth > depthMax) break;
  }

  return depth;
}
```

Pinball: Ray-marching ball-SDF collisions

The function returns the depth along the ray you got during the timestep, which you can test to determine the outcome:

Case 1 (depth ≥ depthMax): No collisions.

- Move remaining step (depthMax).

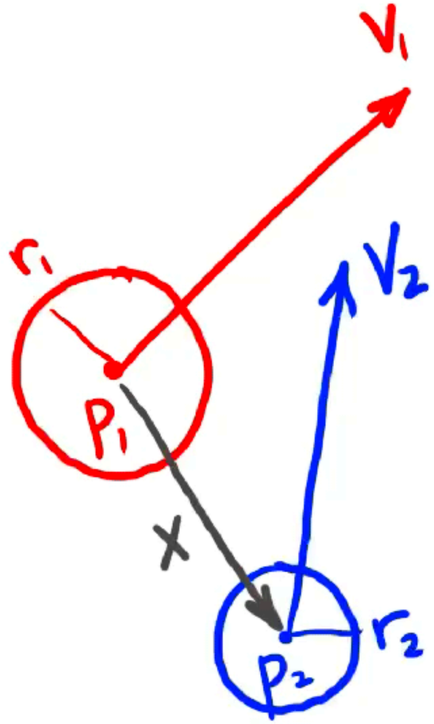
Case 2 (depth < depthMax): Collision found before timestep completed.

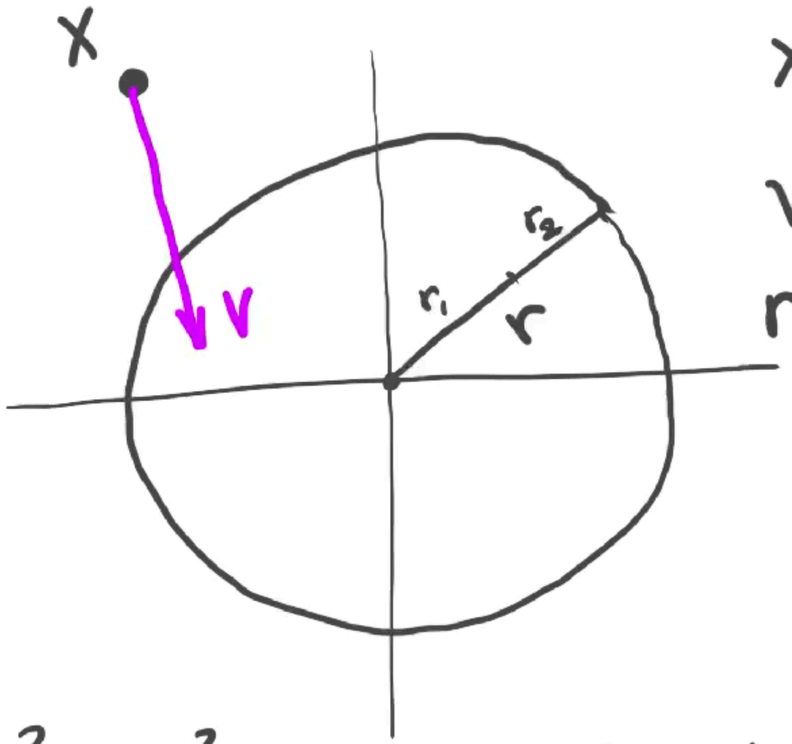
- if(v_n>0) keep moving away from surface.
- if(v_n<0) process collision, i.e., apply an impulse.
- Continue advancing until time Δt elapsed.
- Keep track of your current time.

```
function rayMarchTimestep(p0, vHat, depthMax) {

  let depth = 0;// init depth along ray

  for (let i=0; i<MAX_MARCHING_STEPS; i++) {
    let   p = p0 + depth*vHat; // ball position
    float d = sdf(p); // max safe dist ball can move
    depth += d; // walk along ray “d”
    if (d < PRECISION || depth > depthMax) break;
  }

  return depth;
}
```

Common CCD Tests

CCD Tests: Ball-Ball





$x = P_2 - P_1$

$v = v_2 - v_1$

$r = r_2 + r_1$

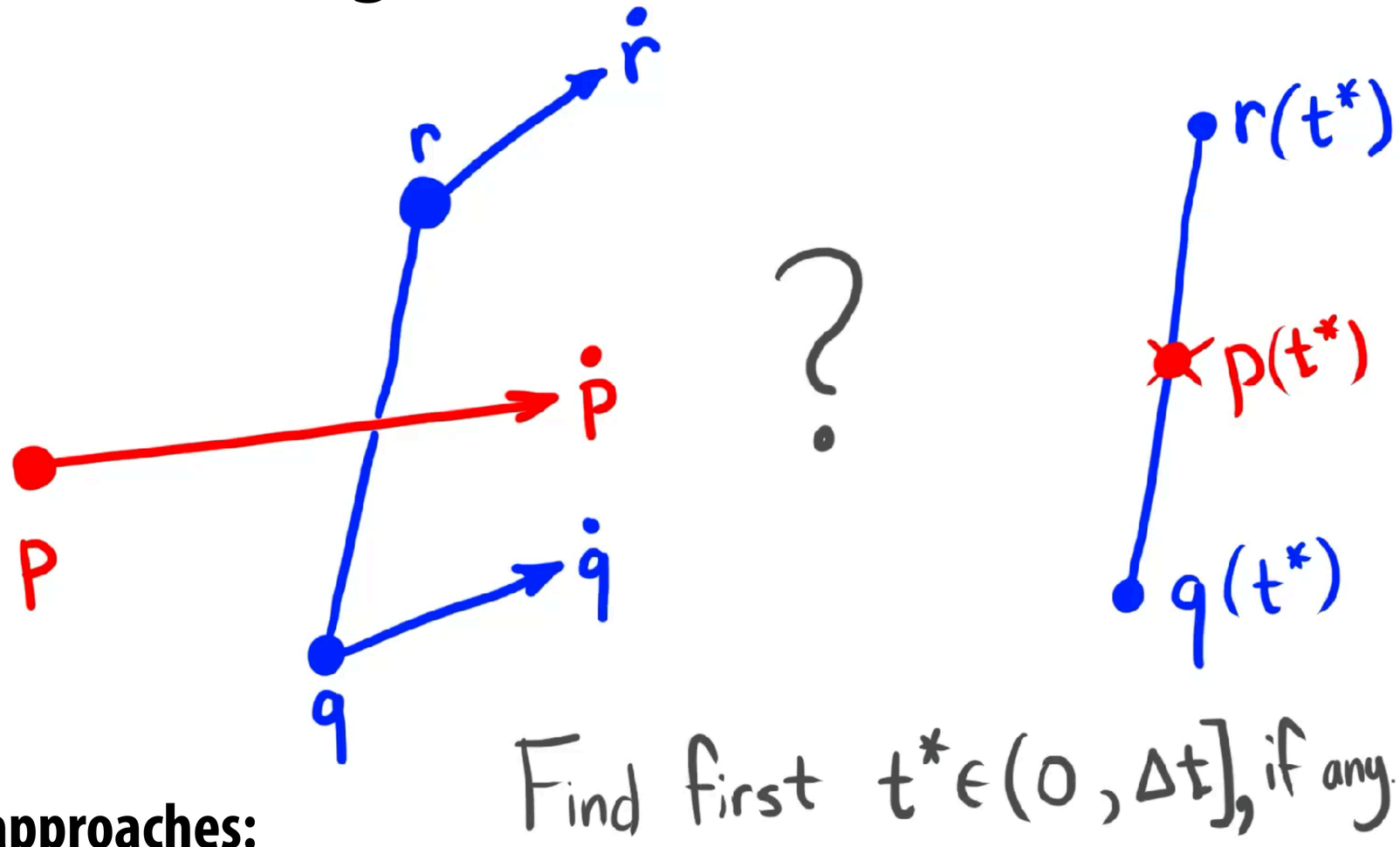
Collision when $\|x(t)\|^2 = r^2$ where $x(t) = x + vt$

$$\Rightarrow r^2 = x(t) \cdot x(t) = (x + tv) \cdot (x + tv)$$

$$\Rightarrow 0 = (v \cdot v)t^2 + (2x \cdot v)t + (x \cdot x - r^2)$$

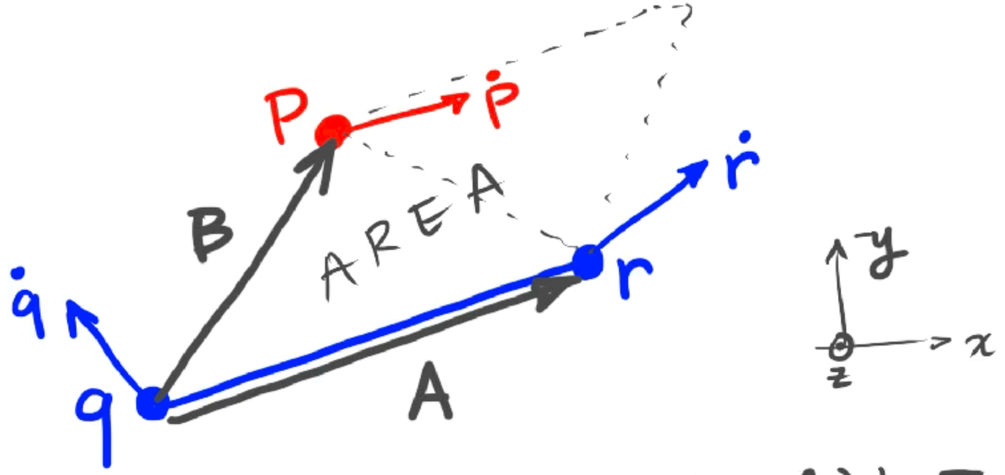
Solve quadratic equation for first root $t^* \in (0, \Delta t]$.

CCD Test: Point-Edge (2D)



- **Different approaches:**
  - **Find time when area of pt-edge triangle is zero, then check if pt on edge.**
  - **Find time when length of ⊥ component is zero, then check if pt on edge.**

Collinearity test



$$A(t) \equiv (r + \dot{r}t) - (q + \dot{q}t) = (r-q) + (\dot{r}-\dot{q})t \equiv a + \dot{a}t$$

$$B(t) \equiv (p + \dot{p}t) - (q + \dot{q}t) = (p-q) + (\dot{p}-\dot{q})t \equiv b + \dot{b}t$$

$$\text{"AREA}(t)\text{"} = \big(A(t) \times B(t)\big)_z \equiv A_x B_y - A_y B_x \equiv A \wedge B$$

$$= (a + \dot{a}t) \wedge (b + \dot{b}t)$$

$$= (\dot{a} \wedge \dot{b})\, t^2 + (\dot{a} \wedge b + a \wedge \dot{b})\, t + (a \wedge b)$$

Find roots of $\text{AREA}(t^*) = 0$.

Find first TOC s.t. $t^* \in (0, \Delta t]$ AND $p(t^*)$ is on $\overline{q(t^*)\, r(t^*)}$.

Robust root-finding methods

- Often need to find the roots of polynomial equations
- Can be numerically tricky for arbitrary computer-generated polynomials from animation
- For example, cloth animations can generate billions of triangle-triangle CCD tests.
- One-in-a-billion bugs are a show stopper.
- Codes must be fast AND highly reliable.