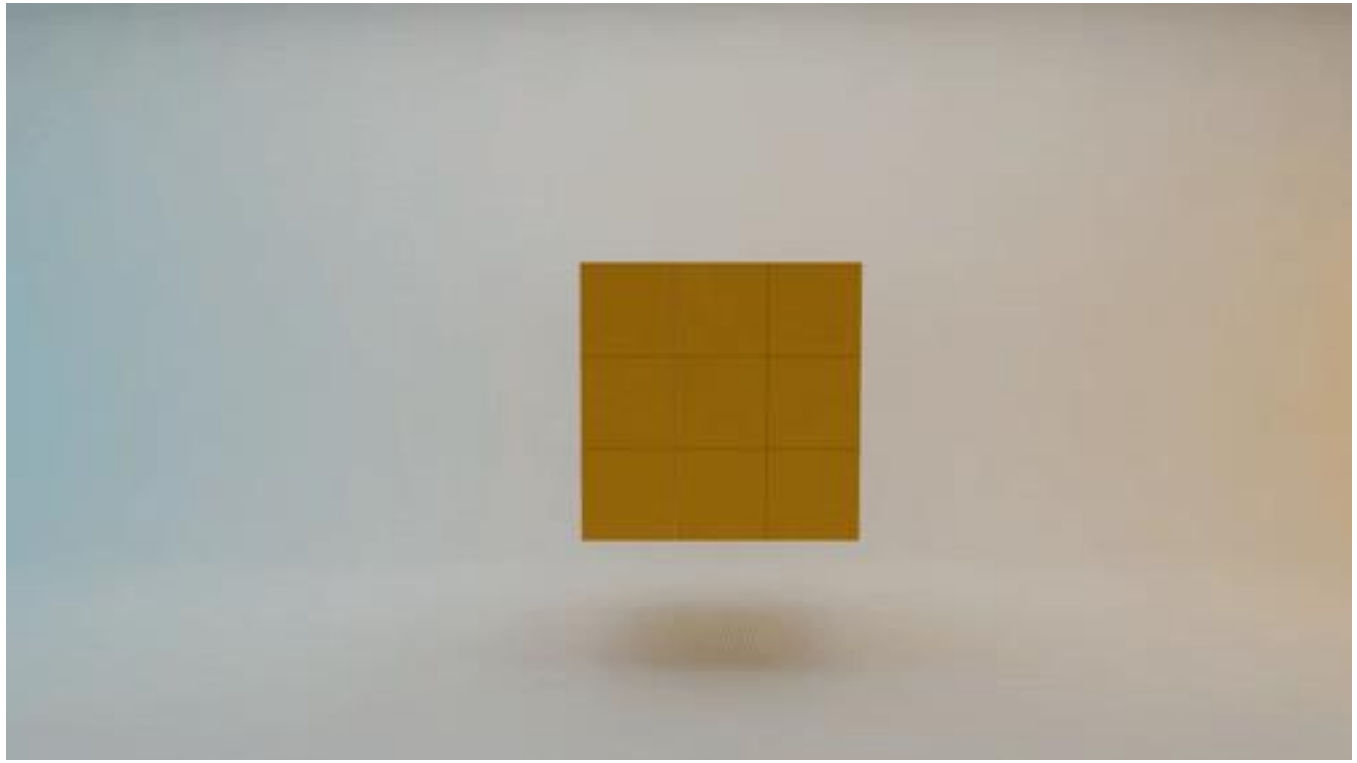
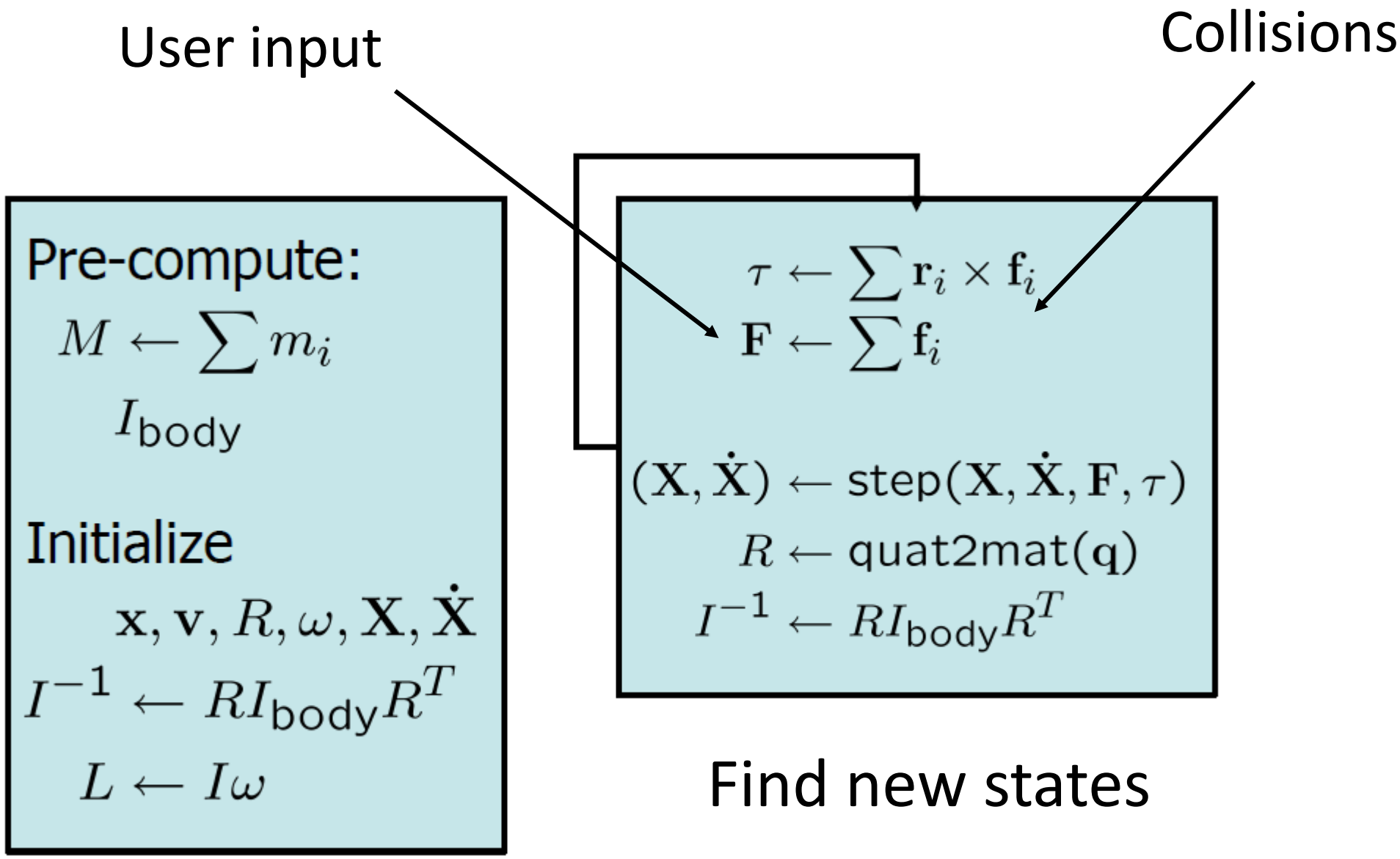
CGP 8

WP

Collision Detection



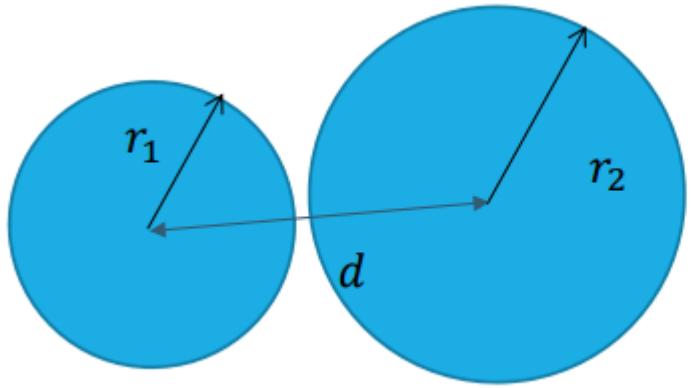

User input
Collisions
Pre-compute:
$M \leftarrow \sum m_i$
I_{body}
Initialize
$\mathbf{x}, \mathbf{v}, R, \omega, \mathbf{X}, \dot{\mathbf{X}}$
$I^{-1} \leftarrow R I_{\text{body}} R^T$
$L \leftarrow I\omega$
$\tau \leftarrow \sum \mathbf{r}_i \times \mathbf{f}_i$
$\mathbf{F} \leftarrow \sum \mathbf{f}_i$
$(\mathbf{X}, \dot{\mathbf{X}}) \leftarrow \text{step}(\mathbf{X}, \dot{\mathbf{X}}, \mathbf{F}, \tau)$
$R \leftarrow \text{quat2mat}(\mathbf{q})$
$I^{-1} \leftarrow R I_{\text{body}} R^T$
Find new states


Find New States
Detect Collisions
Respond to Collisions

Sphere

- Sphere collision check:

- Collision response: conservation of momentum

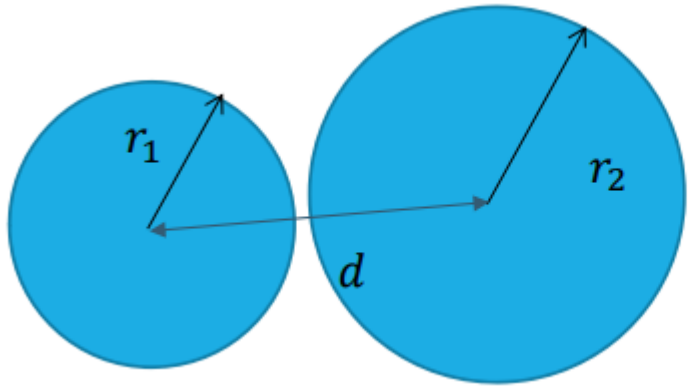


Sphere

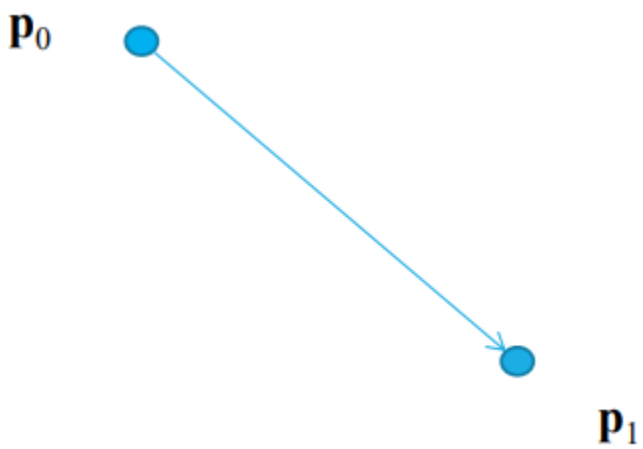
- Sphere collision check:

$$d < r_1 + r_2$$

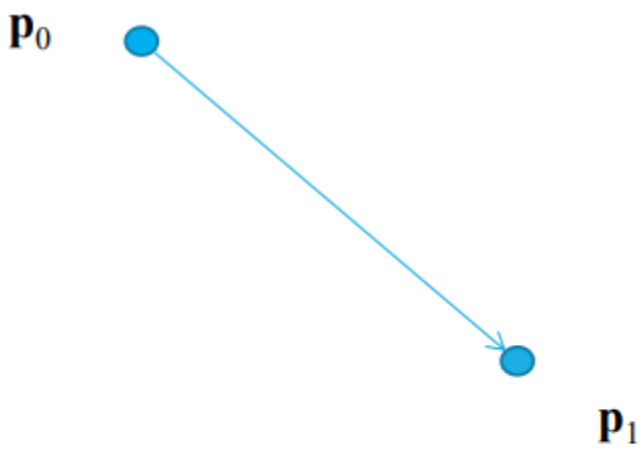
- Collision response: conservation of momentum



Distance: Point to Point



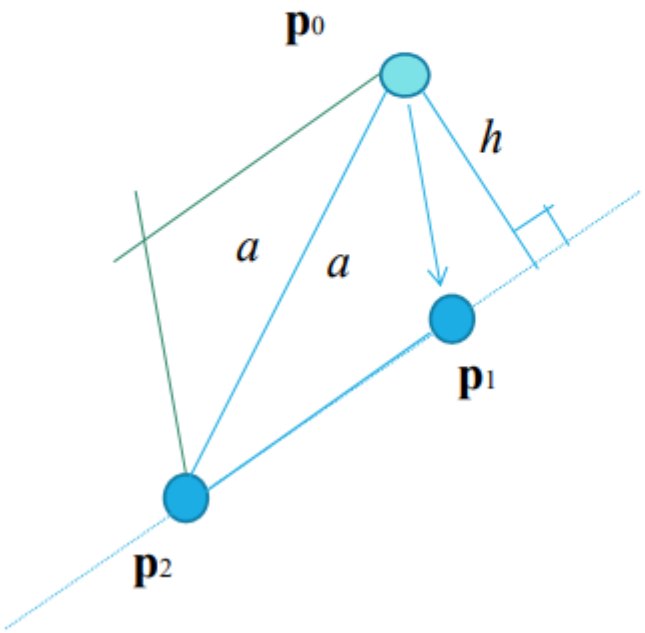
Distance: Point to Point



$$d = |\mathbf{p}_0 - \mathbf{p}_1| = \sqrt{(\mathbf{p}_0.x - \mathbf{p}_1.x)^2 + (\mathbf{p}_0.z - \mathbf{p}_1.y)^2 + (\mathbf{p}_0.z - \mathbf{p}_1.z)^2}$$

Distance: Point to Edge

$$d = \min\left\{\frac{\left|(\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_0-\mathbf{p}_1)\right|}{\left|\mathbf{p}_2-\mathbf{p}_1\right|},\ \left|\mathbf{p}_0-\mathbf{p}_1\right|,\ \left|\mathbf{p}_0-\mathbf{p}_2\right|\right\}$$



Area of the parallelogram A:

$$A = \left|(\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_0-\mathbf{p}_1)\right|$$

The area of the triangle a

$$a = \frac{\left|(\mathbf{p}_2-\mathbf{p}_1)\right|\ h}{2} \quad where \quad a = \frac{A}{2}$$

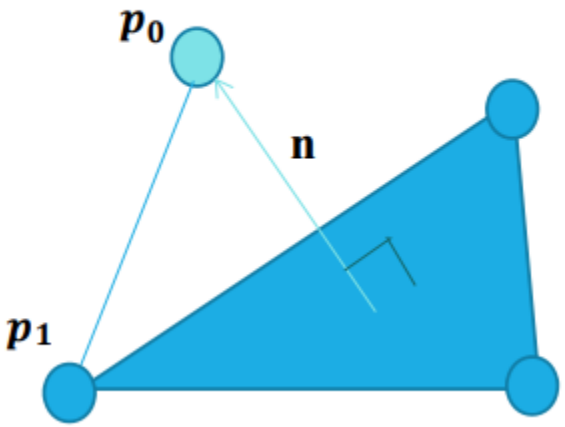
Thus

$$h = \frac{A}{\left|(\mathbf{p}_2-\mathbf{p}_1)\right|} = \frac{\left|(\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_0-\mathbf{p}_1)\right|}{\left|(\mathbf{p}_2-\mathbf{p}_1)\right|}$$

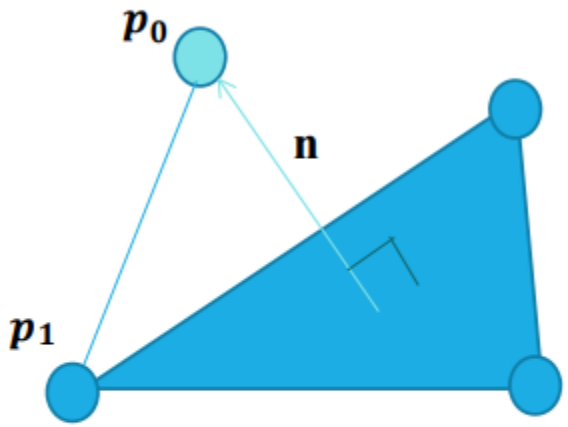
And

$$d = \min\left\{\frac{\left|(\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_0-\mathbf{p}_1)\right|}{\left|\mathbf{p}_2-\mathbf{p}_1\right|},\ \left|\mathbf{p}_0-\mathbf{p}_1\right|,\ \left|\mathbf{p}_0-\mathbf{p}_2\right|\right\}$$

Distance: Point to Face



Distance: Point to Face

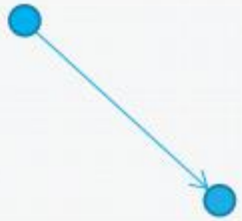


$$d = \min\{(\mathbf{p}_o - \mathbf{p}_1).\hat{\mathbf{n}},\quad \mathbf{p}_o \text{ to } e1, \mathbf{p}_o \text{ to } e2, ... \mathbf{p}_o \text{ to } eN\}$$

Distance Tests

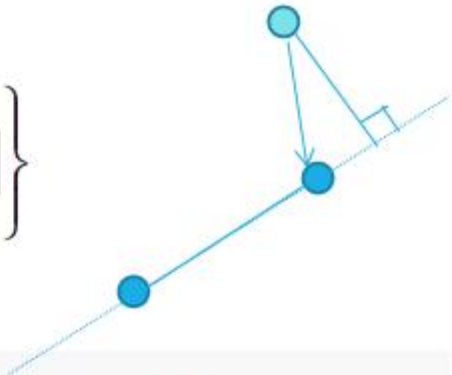
Point to point

$$d = |\mathbf{p}_0 - \mathbf{p}_1|$$



Point-Edge

$$d = \min\left\{\frac{|(\mathbf{p}_2 - \mathbf{p}_1)\times(\mathbf{p}_0 - \mathbf{p}_1)|}{|\mathbf{p}_2 - \mathbf{p}_1|},\ |\mathbf{p}_0 - \mathbf{p}_1|,\ |\mathbf{p}_0 - \mathbf{p}_2|\right\}$$



Point –face

$$d = \min\{(\mathbf{p}_o - \mathbf{p}_1).\hat{\mathbf{n}},\ \ \mathbf{p}_o\ to\ e1, \mathbf{p}_o\ to\ e2, ... \mathbf{p}_o\ to\ eN\}$$

... Edge-edge, edge-face, face-face, object-object

Naïve Collision Detection

```
detect (t_curr, {O_0, ..., O_N-1})
    for t ← t_prev to t_curr in steps of Δt_d
        for each object O_i ∈ {O_0, ..., O_N-1}
            move O_i to its position at time t
        for each object O_i ∈ {O_0, ..., O_N-1}
            for each object O_j ∈ {O_i+1, ..., O_N-1}
            if (O_i penetrates O_j)
              collision occurs at simulation time t
    t_prev ← t_curr
```

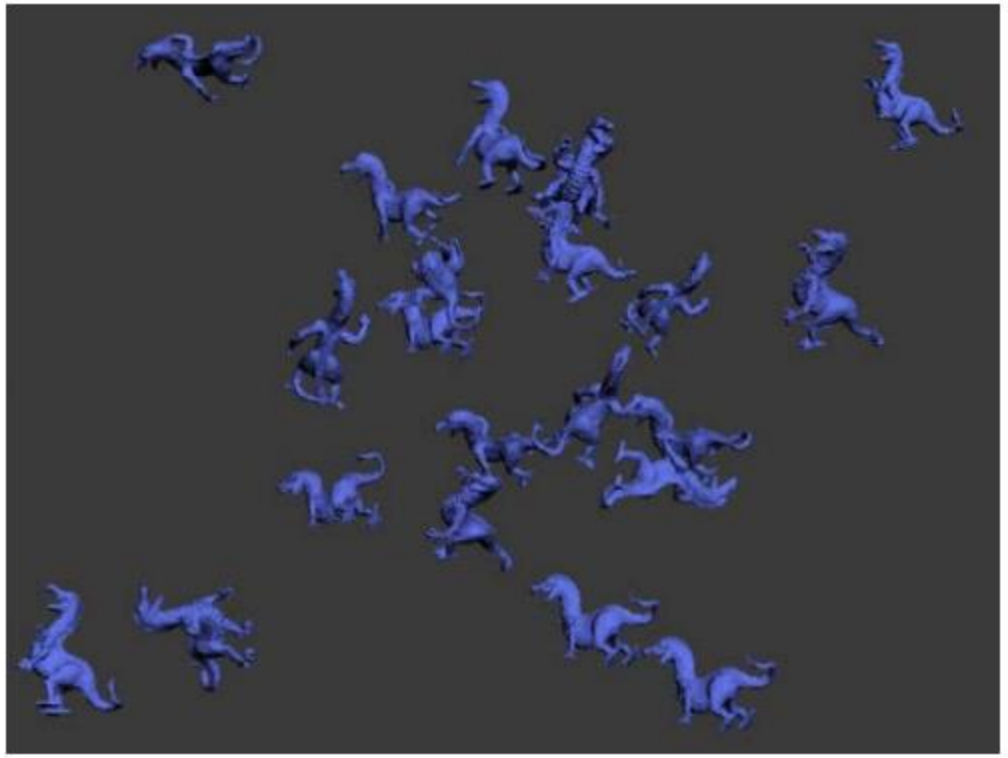
Naïve Collision Detection

```
detect (t_curr, {O_0, ..., O_N-1})
    for t ← t_prev to t_curr in steps of Δt_d
        for each object O_i ∈ {O_0, ..., O_N-1}
            move O_i to its position at time t
        for each object O_i ∈ {O_0, ..., O_N-1}
            for each object O_j ∈ {O_i+1, ..., O_N-1}
            if (O_i penetrates O_j)
                collision occurs at simulation time t
    t_prev ← t_curr
```

Compare ALL Pairs

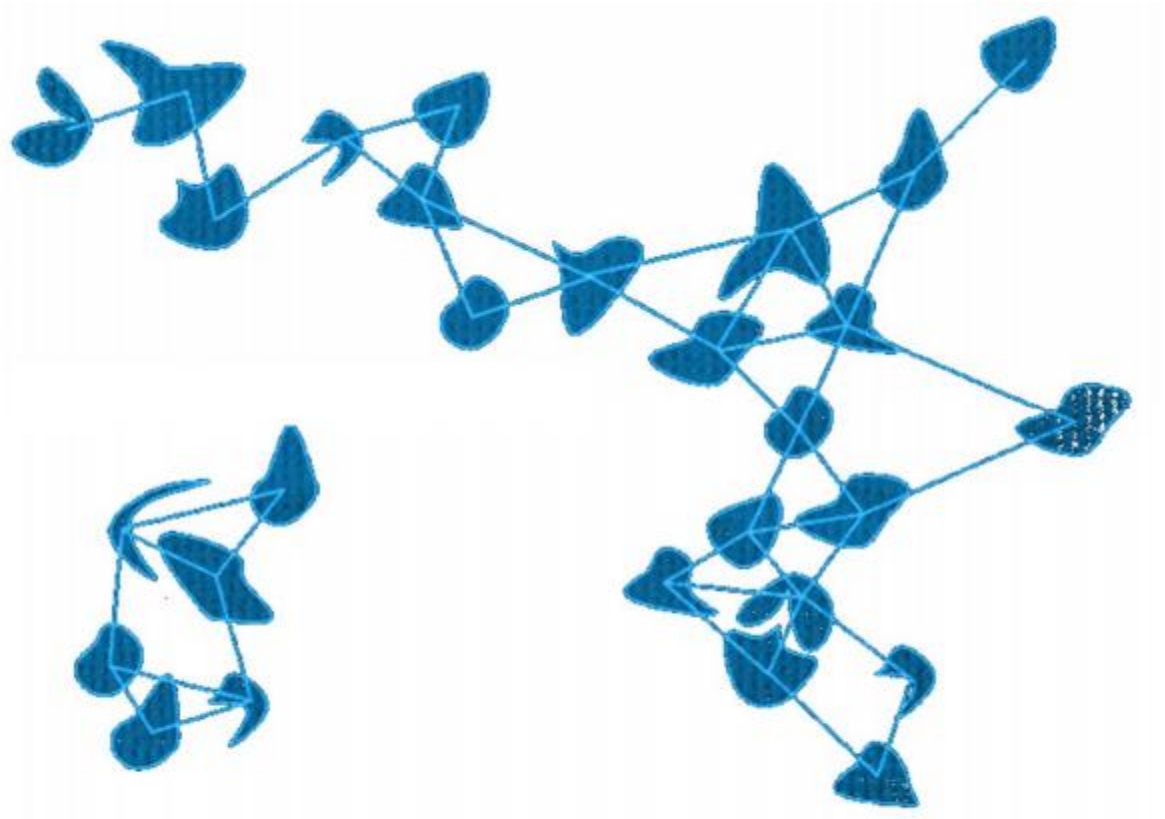
All Pair Inefficiency

Pairwise $\frac{n(n-1)}{2} \approx O(n^2)$ **comparisons**



20 – objects = 190 pair comparisons

Pair Locality



28 Objects = 378 pairs ~ 40 by locality

Naïve Collision Detection

```
detect (t_curr, {O_0, ..., O_N-1})
    for t ← t_prev to t_curr in steps of Δt_d
        for each object O_i ∈ {O_0, ..., O_N-1}
            move O_i to its position at time t
        for each object O_i ∈ {O_0, ..., O_N-1}
            for each object O_j ∈ {O_i+1, ..., O_N-1}
            if (O_i penetrates O_j)
                collision occurs at simulation time t
    t_prev ← t_curr
```

Compare ALL Pairs

Naïve Collision Detection

```
detect (t_curr, {O_0, ..., O_N-1})
    for t ← t_prev to t_curr in steps of Δt_d
        for each object O_i ∈ {O_0, ..., O_N-1}
            move O_i to its position at time t
        for each object O_i ∈ {O_0, ..., O_N-1}
            for each object O_j ∈ {O_i+1, ..., O_N-1}
            if (O_i penetrates O_j)
                collision occurs at simulation time t
    t_prev ← t_curr
```

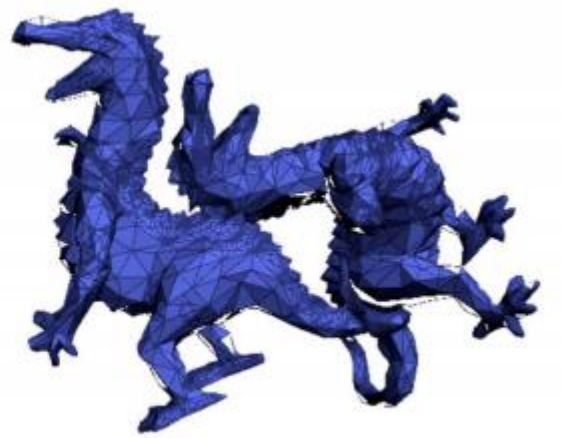
Compare ALL Pairs

Pair Processing

Pair Processing Inefficiency

- Even for just two pairs there are significant problems:

2 objects (100 polygons) ~ 10000 polygon pairs to check



Naïve Collision Detection

```
detect (t_curr, {O_0, ..., O_N-1})
    for t ← t_prev to t_curr in steps of Δt_d
        for each object O_i ∈ {O_0, ..., O_N-1}
            move O_i to its position at time t
        for each object O_i ∈ {O_0, ..., O_N-1}
            for each object O_j ∈ {O_i+1, ..., O_N-1}
            if (O_i penetrates O_j)
                collision occurs at simulation time t
    t_prev ← t_curr
```

Compare ALL Pairs

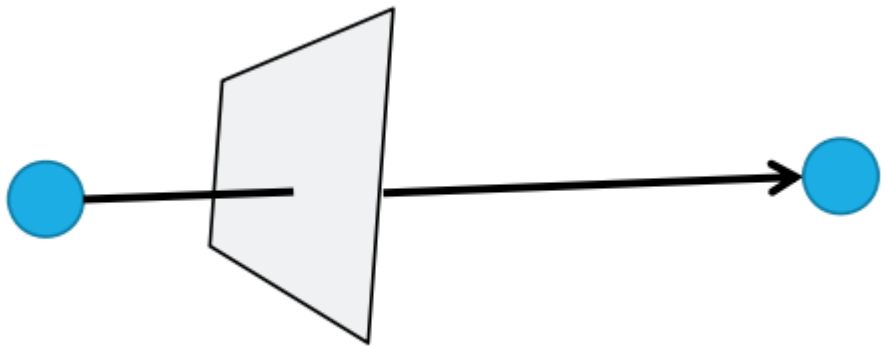
Pair Processing

Fixed Time-Step Problem

Naive collision detection algorithm

```
detect(t_curr, {O_0, ..., O_N-1})
    for t ← t_prev to t_curr in steps of Δt_d
        for each object O_i ∈ {O_0, ..., O_N-1}
            move O_i to its position at time t
        for each object O_i ∈ {O_0, ..., O_N-1}
            for each object O_j ∈ {O_i+1, ..., O_N-1}
            if (O_i penetrates O_j)
              collision occurs at simulation time t
    t_prev ← t_curr
```

Discrete time step

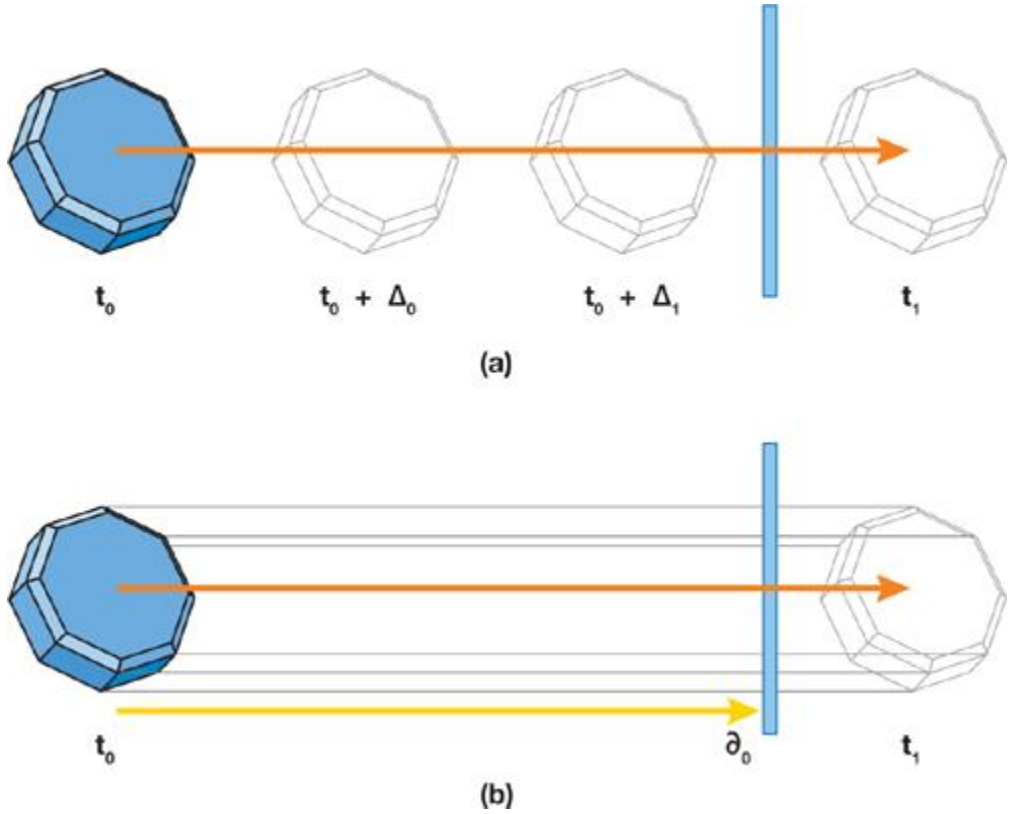


Fixed Time-Step Problem

- Position and orientation of bodies are updated at each time-step
  - Interactions are only handled at discrete time-steps

- Simulation time is incremented by fixed amount Δt
  - Large Δt reduces accuracy of collision detection
  - Smaller Δt reduces efficiency of simulation

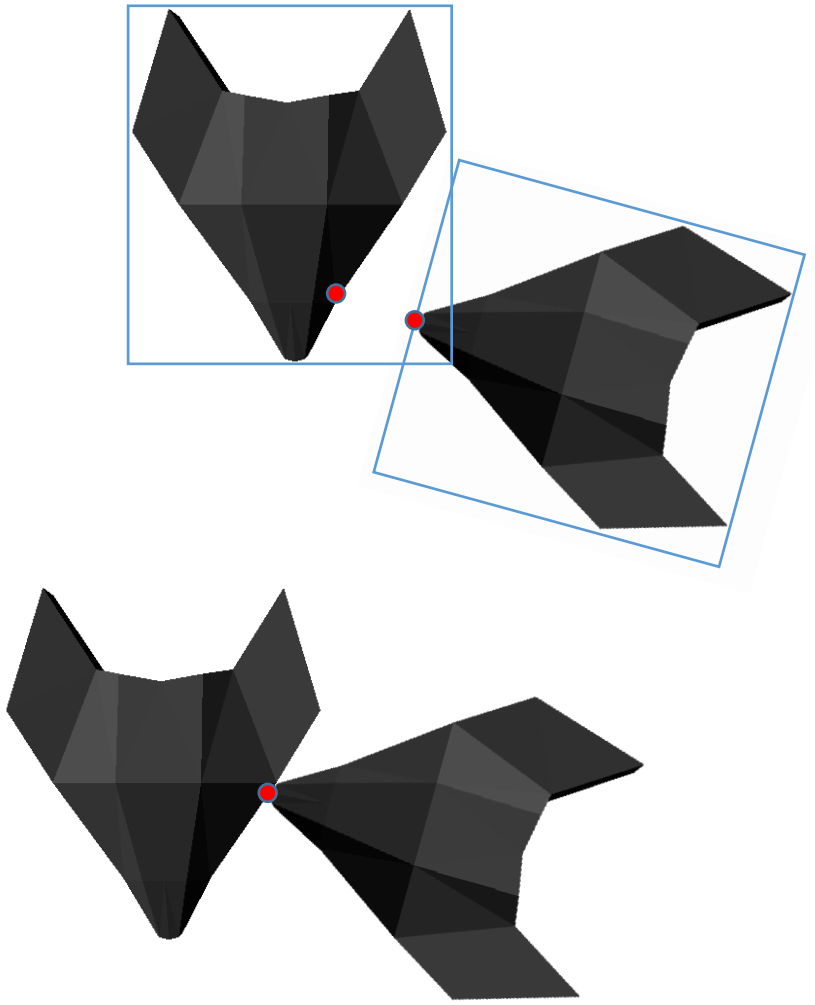
Continuous collision detection



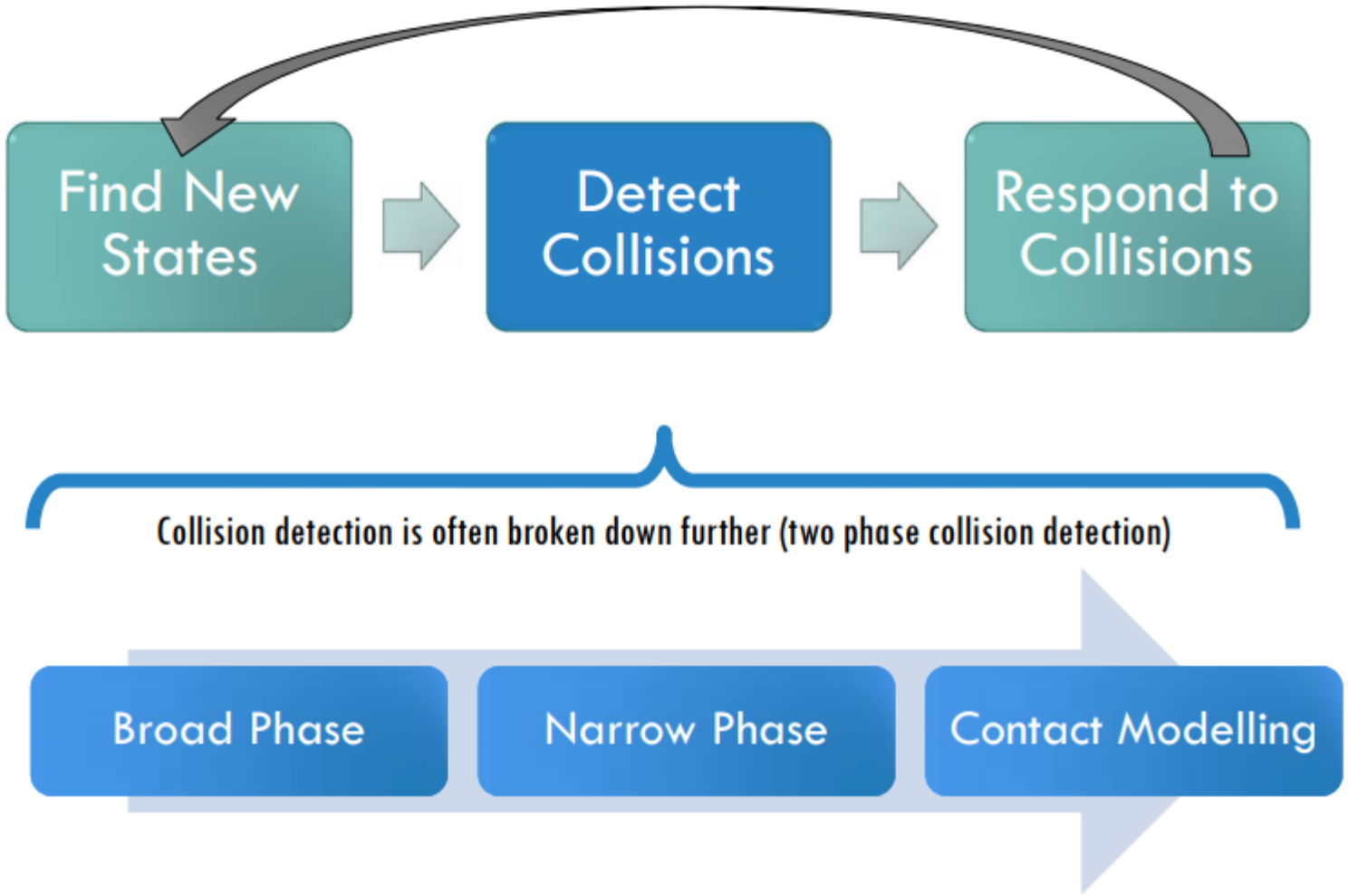
Nvidia GPU Gems 3 Ch. 33

Two-Phase Collision Detection

- Addressing the issues:
  - **Broad Phase**: use coarse *bounding volumes* to represent objects and *cull* pairs who overlap. "*Sweep and Prune*" exploits between–frame coherence to achieve almost O(n) complexity
  - **Narrow Phase**: use polygon model to determine actual points of contact, but *cull* areas of objects that are not in contact

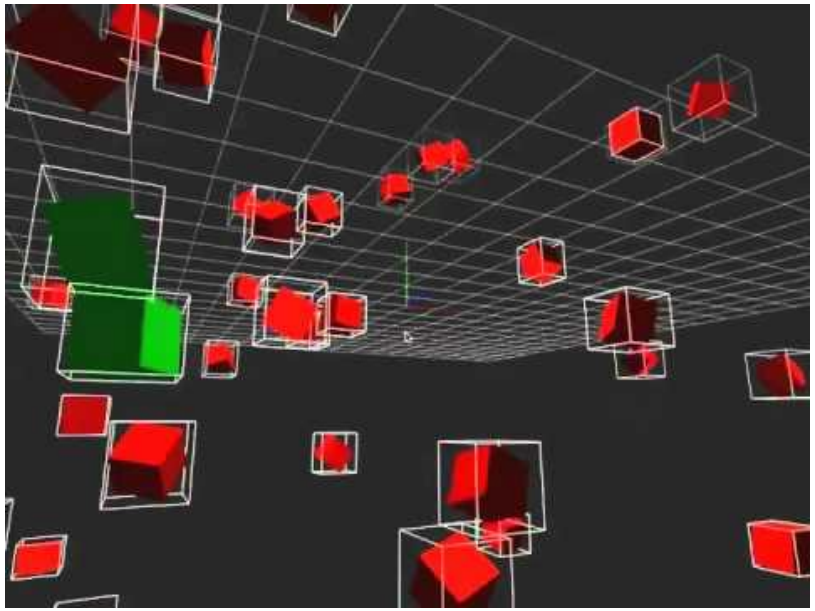


Basic Simulation Loop

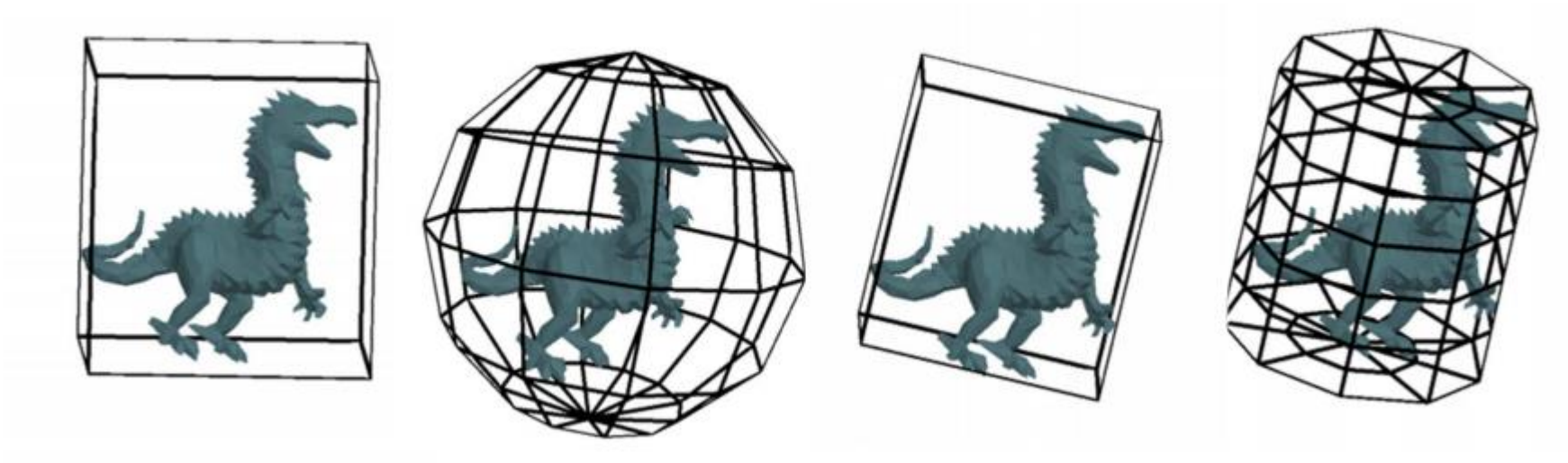


Broad-Phase Collision Detection

- Quickly prune away pairs of objects from more detailed collision processing
- Faster than $O(n^2)$ performance achieved by exploiting features in animation data:
  - Locality
  - Coherency
- Typical Instruments:
  - **Bounding volumes**
  - **Spatial subdivision**
  - **Sweep and Prune**

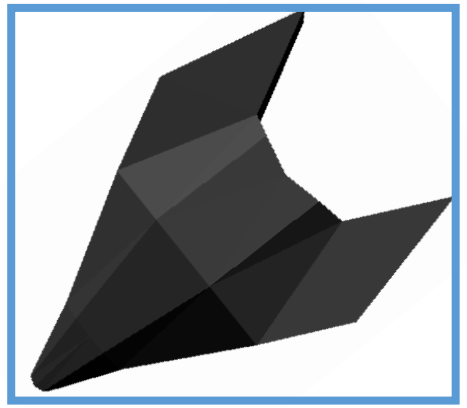


Bounding Volumes



Axis-Aligned Bounding Box (AABB)

- Box with edges that align to the axes of coordinate system (position limits in each dimension)
- Find minimum and maximum coordinate values in x, y and z directions
- Pros
  - Computationally efficient
- Cons
  - Unsatisfying fill efficiency
  - Not invariant to rotation (requires dynamic update)



Bounding Sphere

- The smallest sphere that encloses object (center, radius)
- Find minimum enclosing sphere
  - Take center of AABB; furthest vertex is radius
  - Ritter's or Welzl's algorithm
- Pros
  - Invariant to rotation
  - Computationally efficient
- Cons
  - Long or flat objects are not well fit

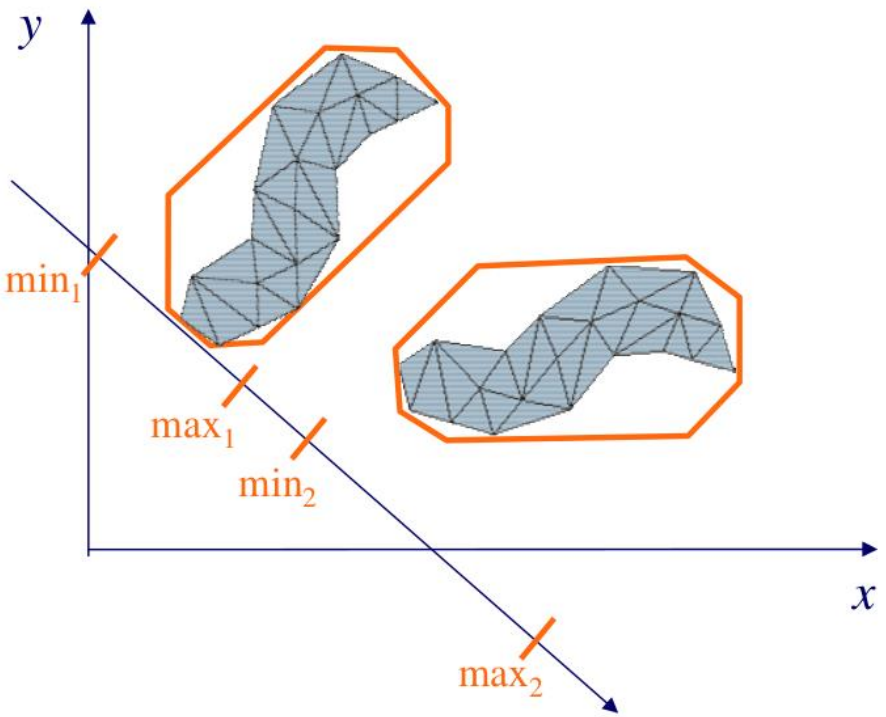
Oriented Bounding Box (OBB)

- Optimally fit box around object (position, orientation, extents)
- Create manually or used PCA based fitting
- Pros
  - Invariant to rotation
  - Tighter bounds than AABB and sphere
- Cons
  - Computationally more expensive to generate
  - Harder to implement

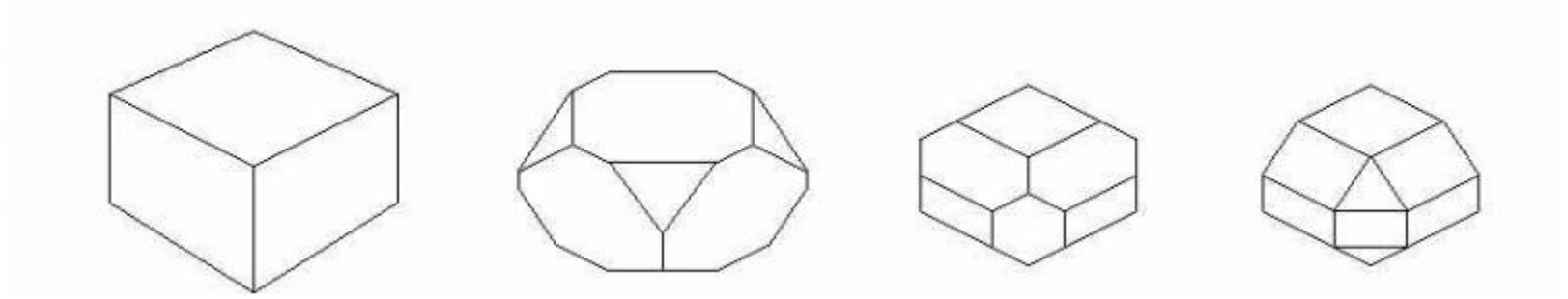
Discrete Oriented Polytopes (k-DOP)

- Generalization of AABBs defined by k hyperplanes with normal in discrete directions (n_k: $n_{k,j} \epsilon \{0, \pm 1\}$)
- K-DOP is defined by k/2 pairs of min, max values in k directions.

- Two k-DOPs do not overlap, if the intervals in one direction do not overlap.



Different k-DOPs



6-DOP

=AABB

14-DOP

(corners)

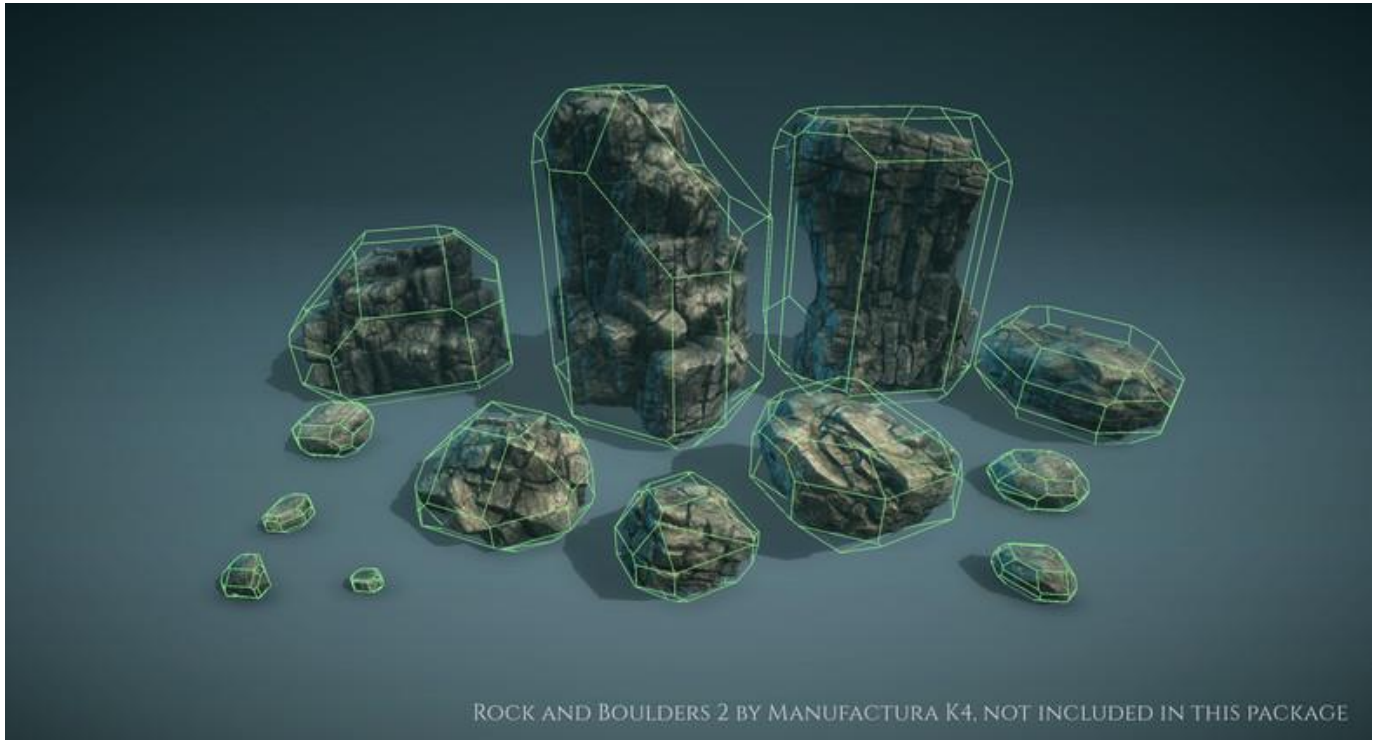
18-DOP

(edges)

26-DOP

(edges and corners)

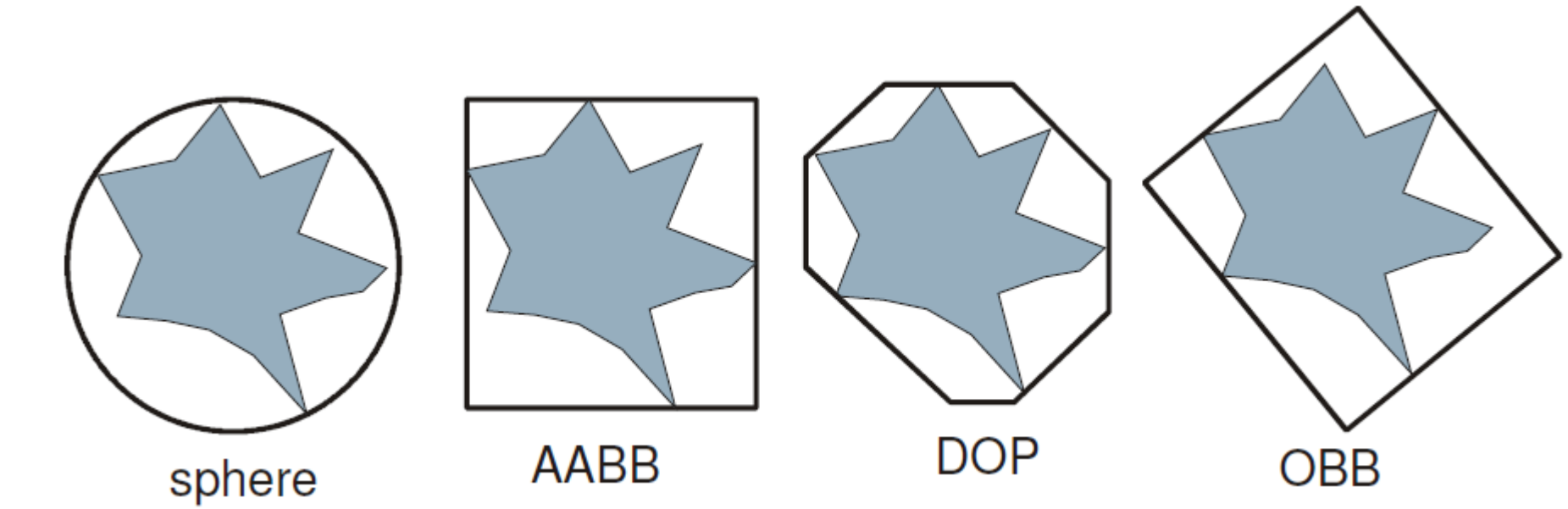
k-DOP



Unity k-DOPs

Broad Phase Objects

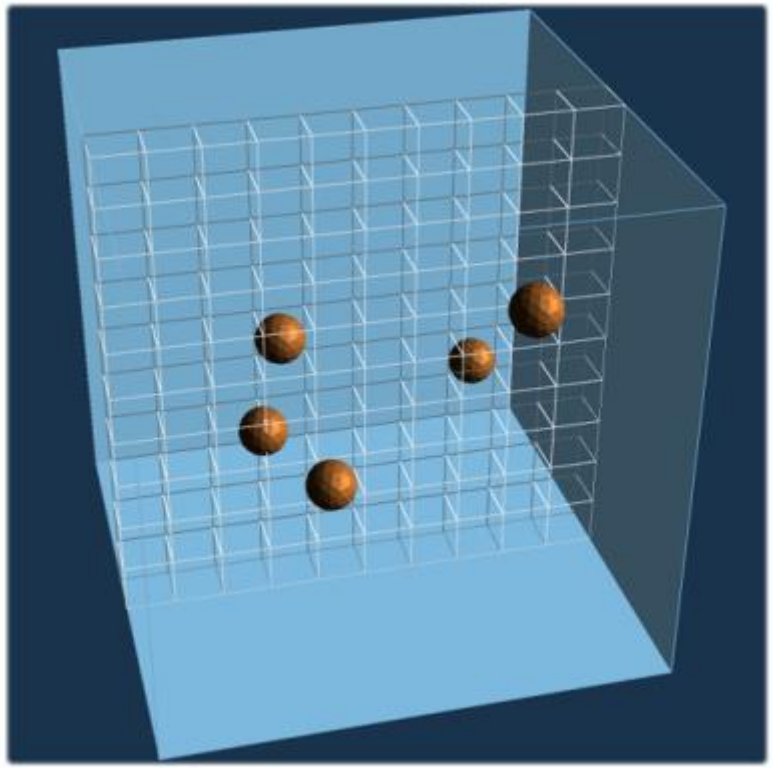
Better approximation, higher build and update costs



Smaller computational costs for overlap tests

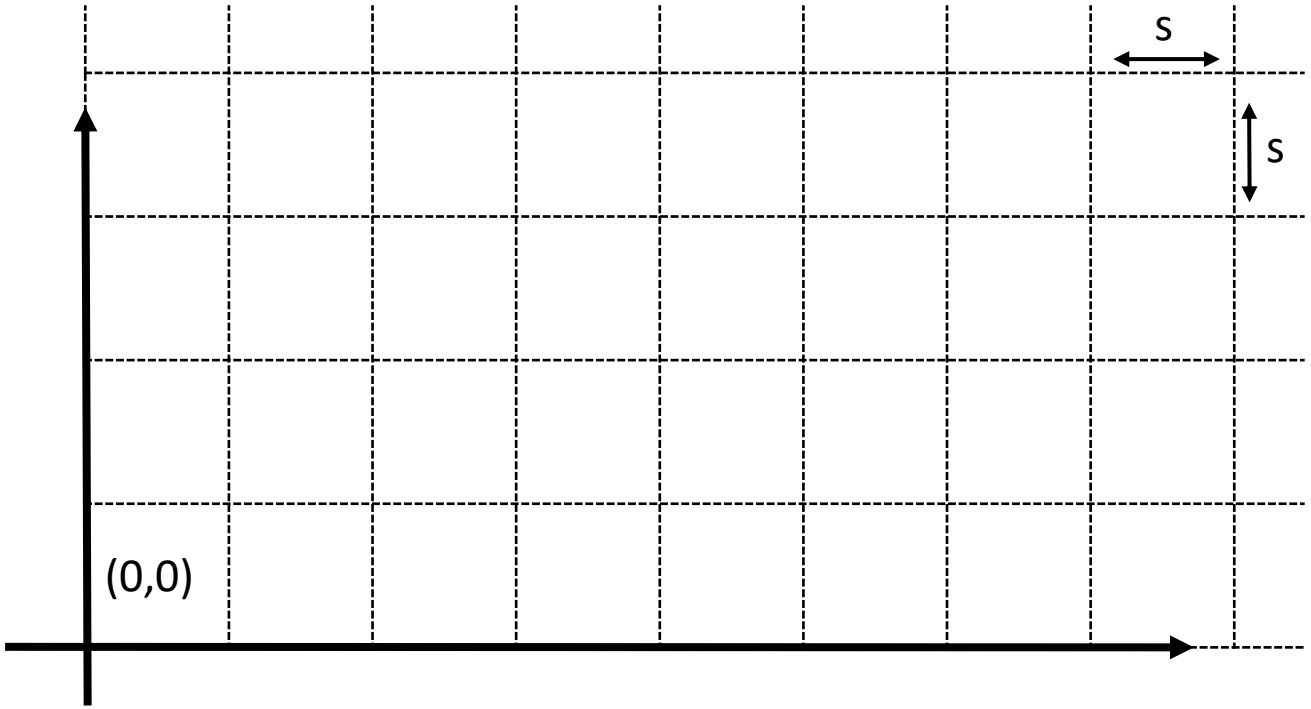
Spatial Subdivision

- E.g. Grid method
  - Divide scene into uniform cells
  - Each object keeps a record of grid cells that it overlaps with
  - Only perform pair-wise collision tests with objects in own or neighboring cells
- Define **topology** of objects



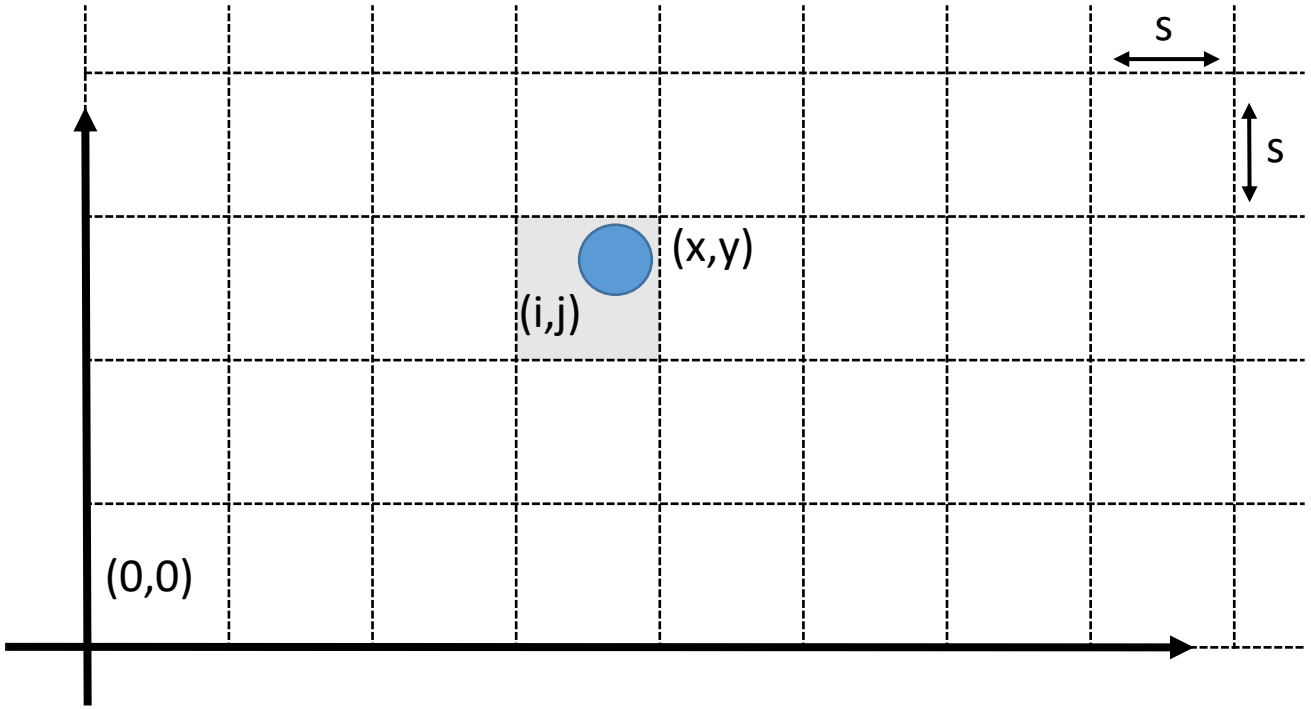
Uniform Grid

- Define a uniform grid with cell size s



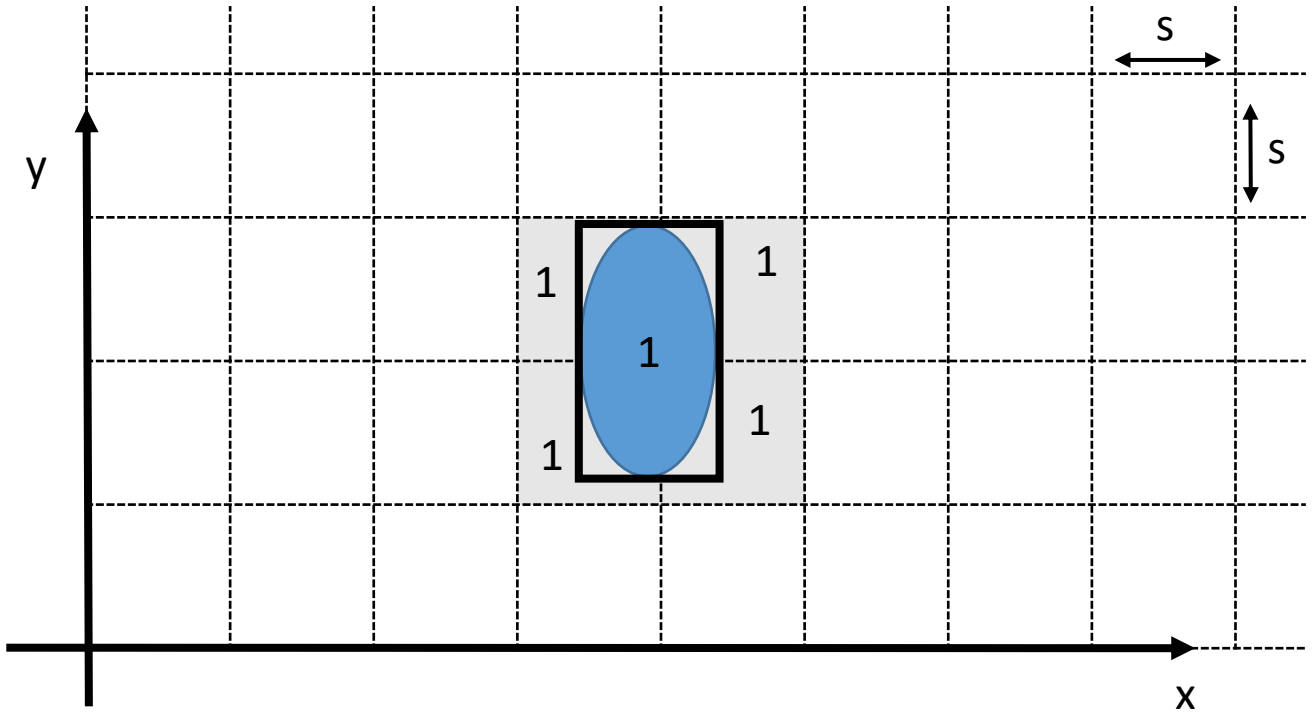
Uniform Grid

- Define a uniform grid with cell size s
- For each point p = (x,y,z) we can find corresponding cell c = (i,j,k) = T(p)
- Tiling function T(p) = (⌊x/s⌋, ⌊y/s⌋, ⌊z/s⌋)



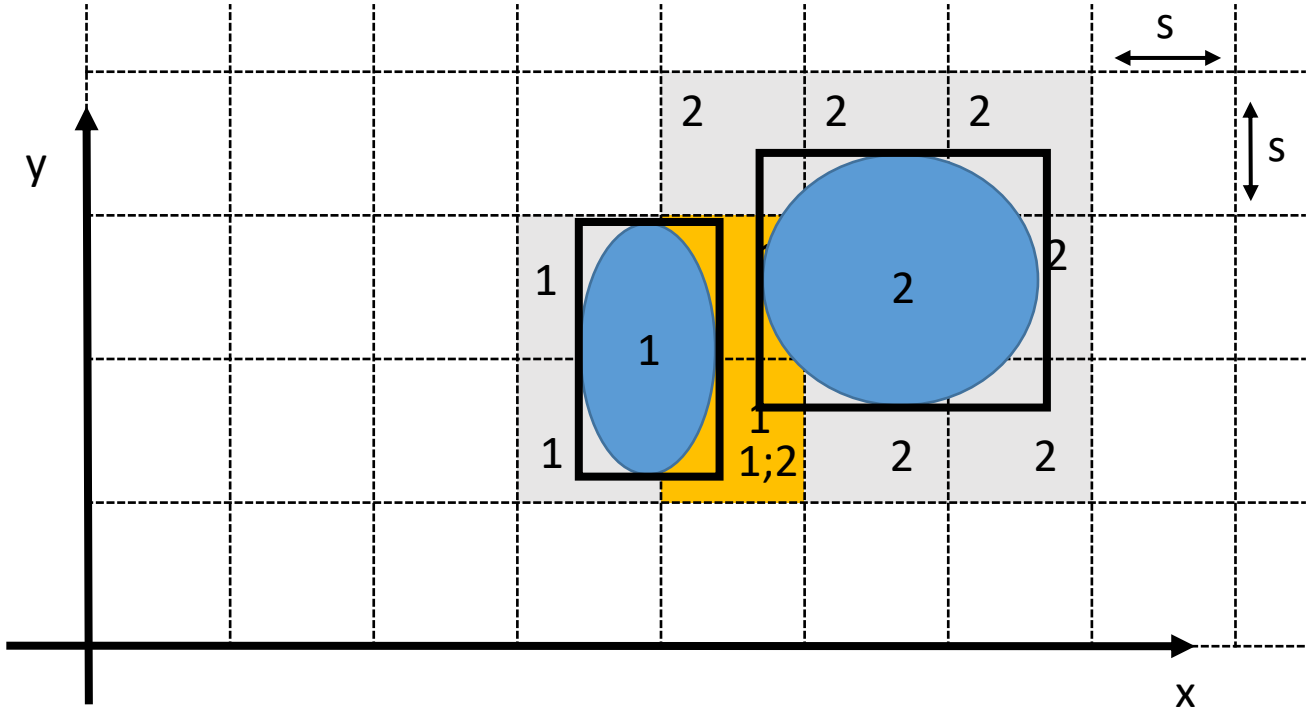
Uniform Grid

- Insert object (ID=1) into grid and store it's ID into overlapping cells based on its AABB



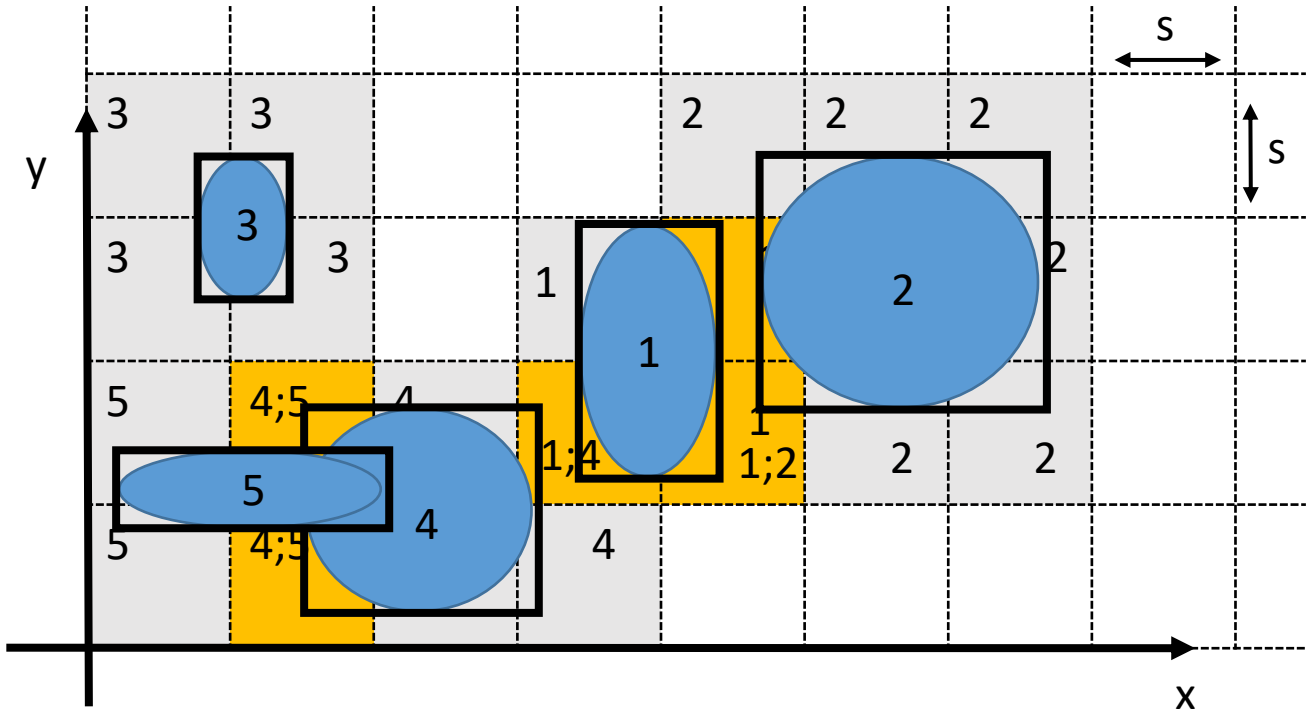
Uniform Grid

- Insert object (ID=1) into grid and store it's ID into overlapping cells based on its AABB
- Insert object (ID=2) into grid …



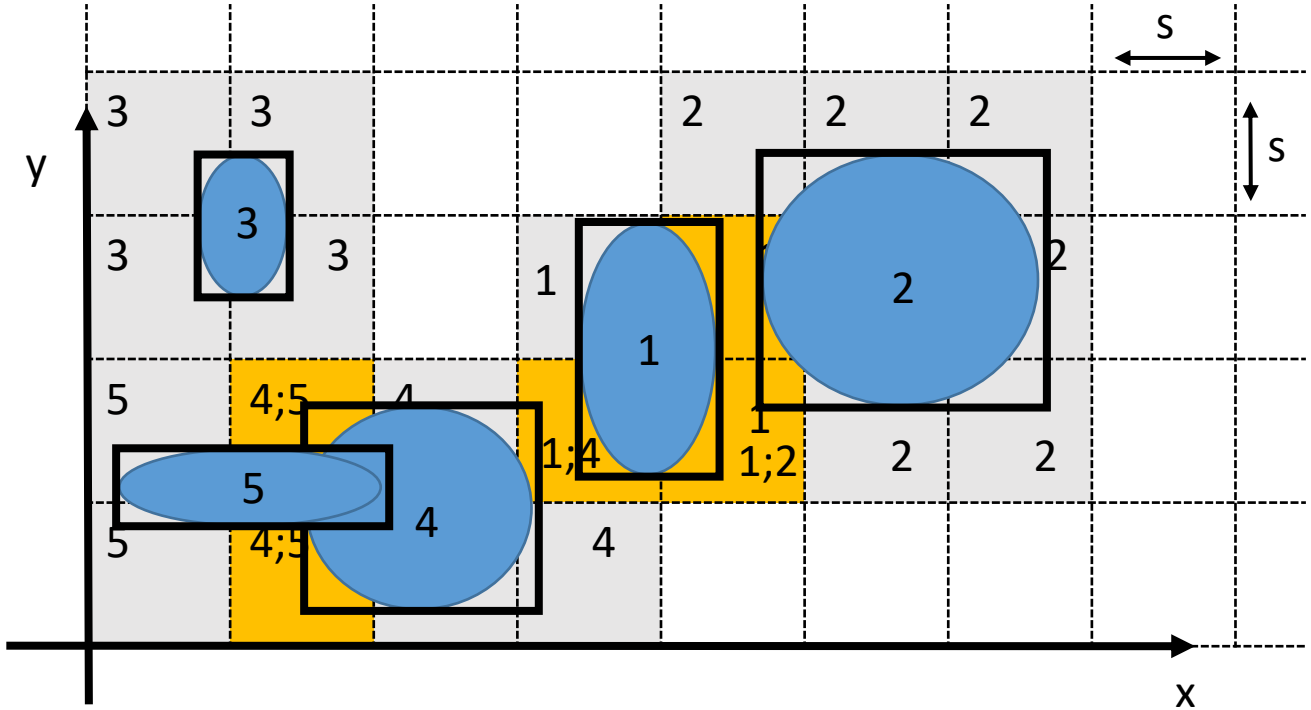
Uniform Grid

- Insert all objects into grid and store IDs into cells
  - Orange colored cells contain several IDs



Uniform Grid

- Insert all objects into grid and store IDs into cells
  - Orange colored cells contain several IDs - define colliding pairs
- Colliding pairs: (1-2),(1-4),(4-5)



Uniform Grid – Add Box

- To add a new object “A” into grid
- Obtain AABB(A) = (A_{x-}, A_{y-}, A_{z-}, A_{x+} , A_{y+} , A_{z+}) of “A”
- Calculate Cell(A) = (A_{i-}, A_{j-}, A_{k-}, A_{i+}, A_{j+}, A_{k+})
- For each cell within (A_{i-}, A_{j-}, A_{k-}) and (A_{i+}, A_{j+}, A_{k+})
  - For each ID stored in the cell create pair (ID_x , ID)
  - Add ID of object to the list of IDs (check duplicates)

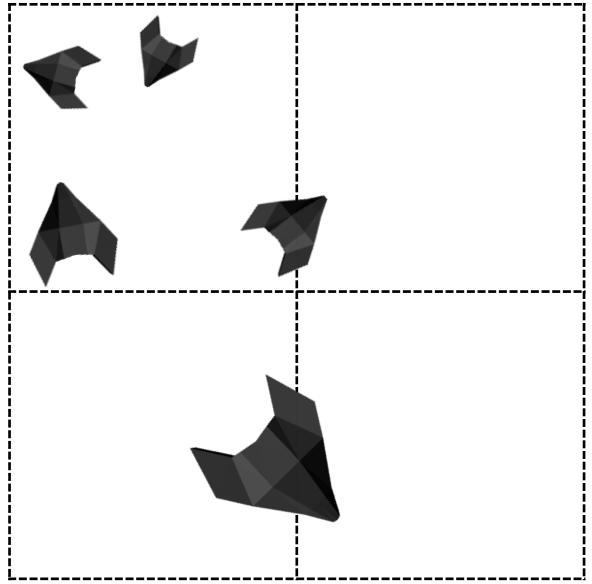
Uniform Grid – Remove Box

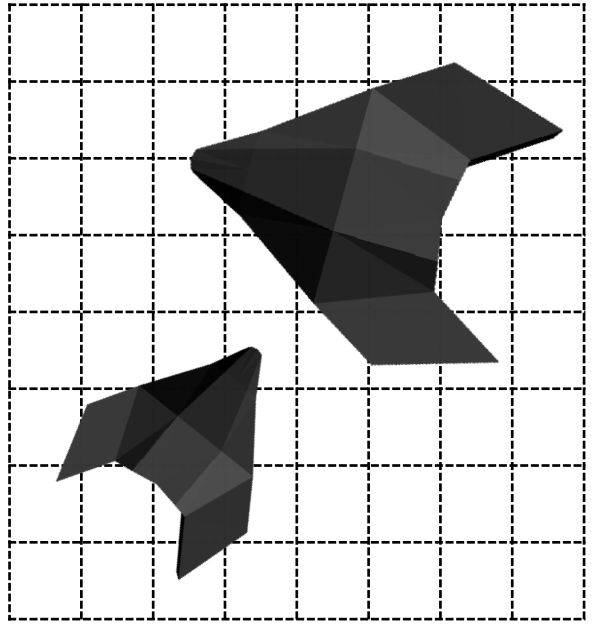
- To remove an existing object from grid
- Obtain AABB(A) = (A_{x-}, A_{y-}, A_{z-}, A_{x+} , A_{y+} , A_{z+}) of “A”
- Calculate Cell(A) = (A_{i-}, A_{j-}, A_{k-}, A_{i+}, A_{j+}, A_{k+})
- For each cell within (A_{i-}, A_{j-}, A_{k-}) and (A_{i+}, A_{j+}, A_{k+})
  - For each ID stored in the cell remove pair (ID_x , ID)
  - Remove ID of object to the list of IDs

Uniform Grid – Update Box

- When an object has moved the corresponding cells have to be updated
- Simple approach: call RemoveBox than AddBox
- Alternatively, find specific cells where we need to add/remove IDs

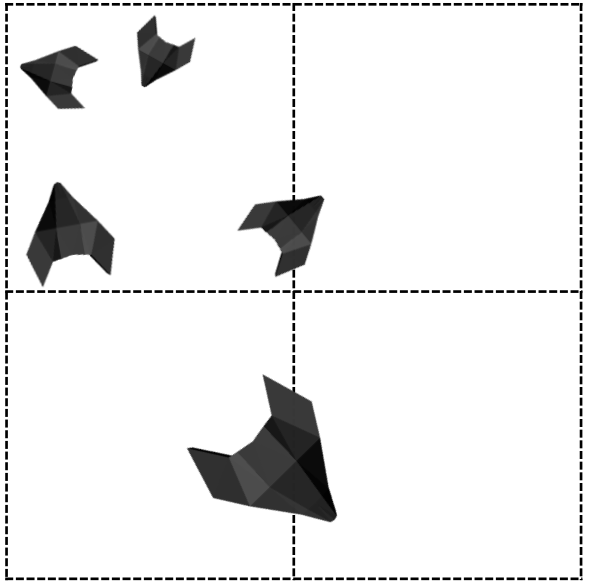
Uniform Grids - Considerations



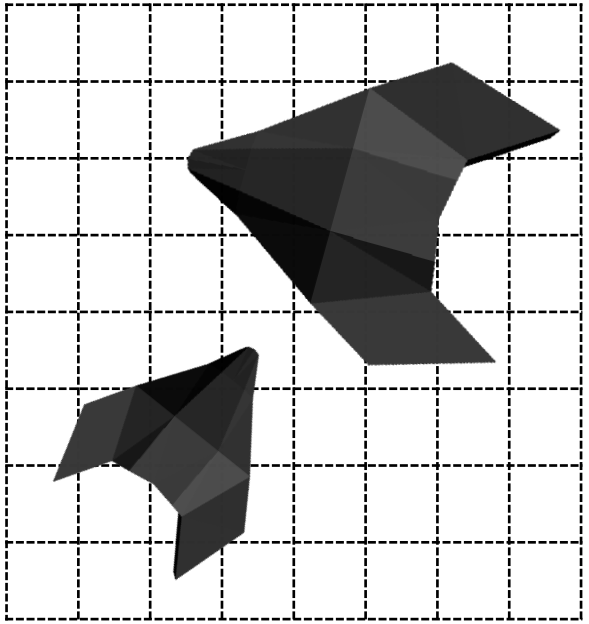


Uniform Grids - Considerations

- If size is too big in relations to objects: too many false positives in narrow phase

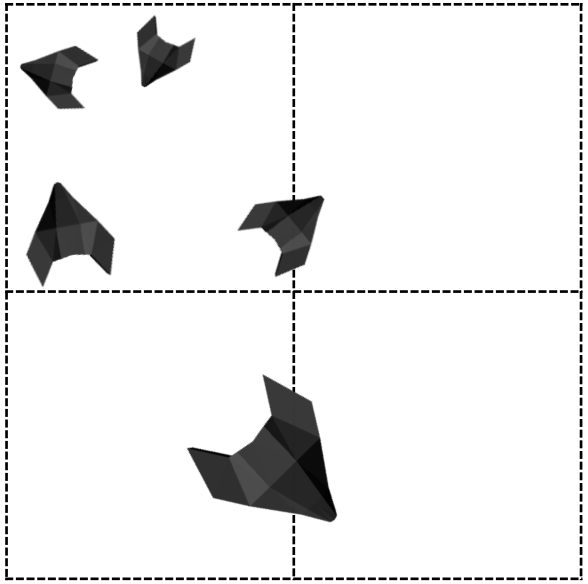


- If grid size too small: too much work in broad phase

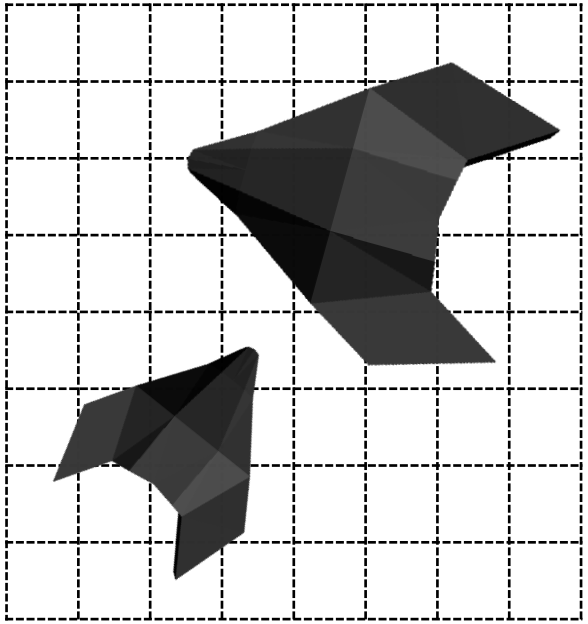


Uniform Grids - Considerations

- If size is too big in relations to objects: too many false positives in narrow phase



- If grid size too small: too much work in broad phase



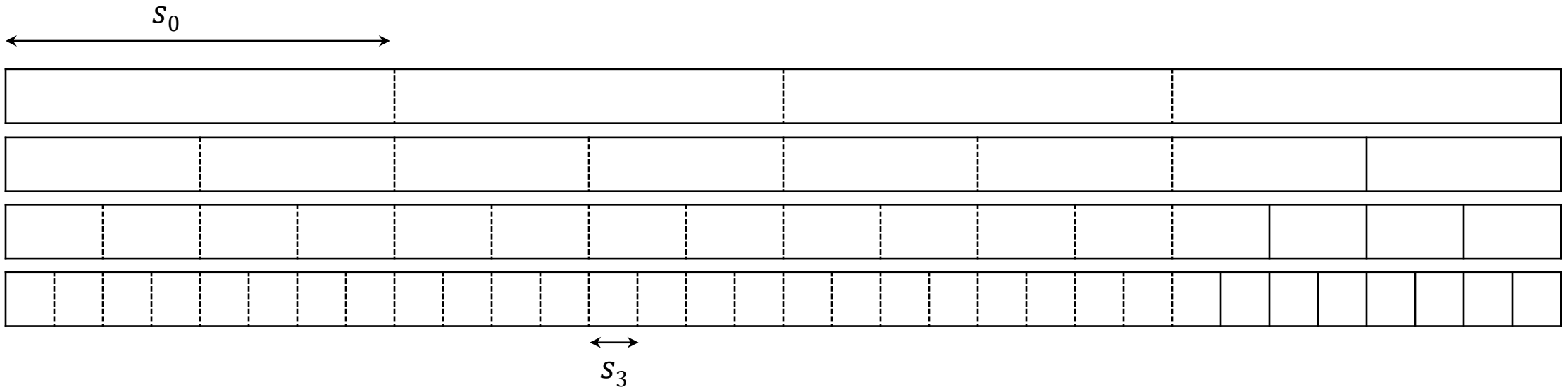
- In practice: size of cell a little bit larger than longest edge of largest AABB

Uniform Grid - Summary

- Pros
  - Simple algorithm – easy to implement
  - Fast in special cases – same-size dynamic objects
- Cons
  - Finding optimal grid size → problem with large vs small dynamic objects
  - Large 3D grid → large amount of memory
  - Slow grid update for large objects

Hierarchical Uniform Grid

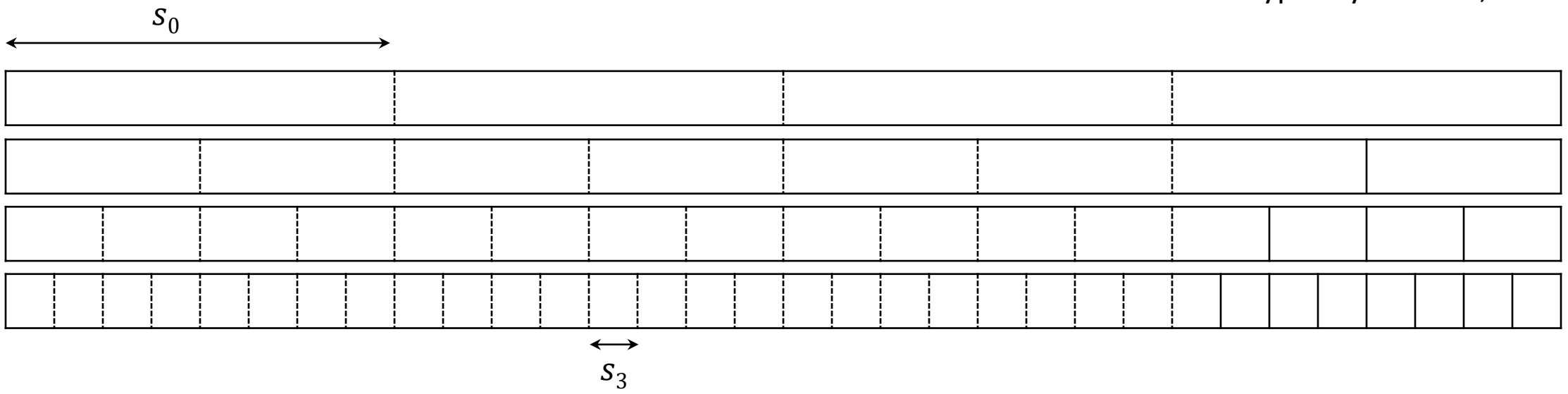
- Suppose 4 uniform grids with 2^k resolutions
  - Grid-0: cell size $s_0 = 1/2^0 = 1.000$
  - Grid-1: cell size $s_1 = 1/2^1 = 0.500$
  - Grid-2: cell size $s_2 = 1/2^2 = 0.250$
  - Grid-3: cell size $s_3 = 1/2^3 = 0.125$



Hierarchical Uniform Grid

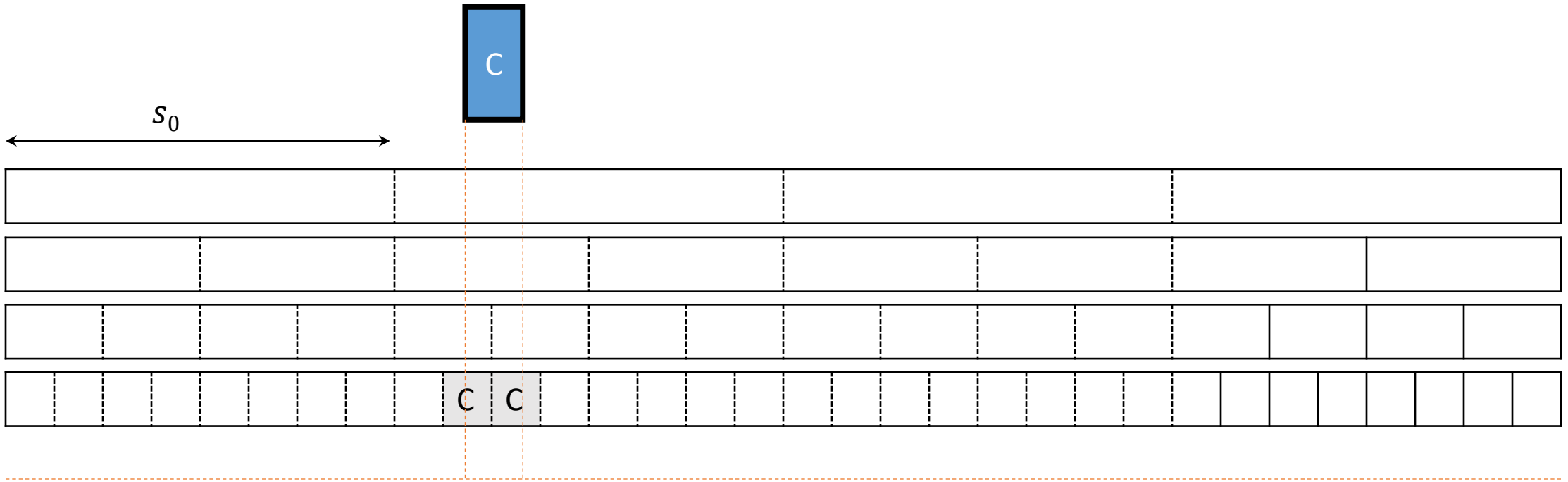
- Find resolution of object “C”: Res(C) = 3
  - Cell sizes in grids: S = (s_0 ,s_1 , … ,s_k)
  - Object box: AABB(C) = (C_{x-}, C_{y-}, C_{z-}, C_{x+} , C_{y+} , C_{z+})
  - Object size: Size(C) = max(C_{x+}- C_{x-}, C_{y+}- C_{y-} , C_{z+} - C_{z-})
  - Object resolution**: Res(C) = i <=> a $\leq$ (Size(C)/s_i) $\leq$ b**

Typically: a = 0.5 ; b = 1



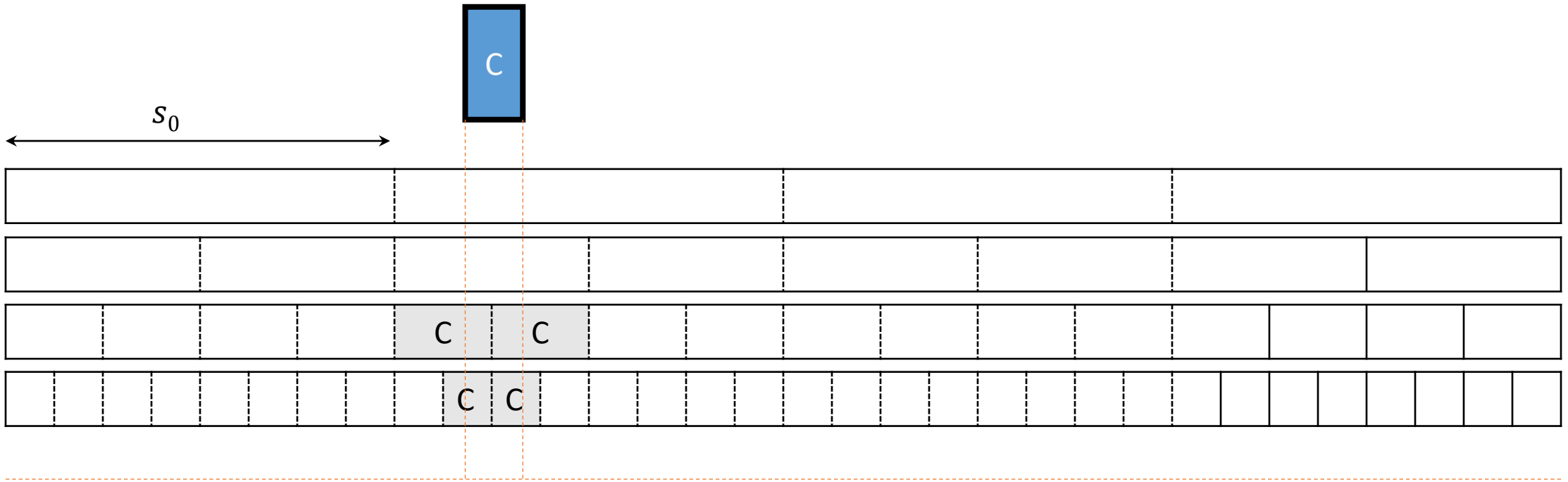
Hierarchical Uniform Grid

- Insert “C” into grid-3



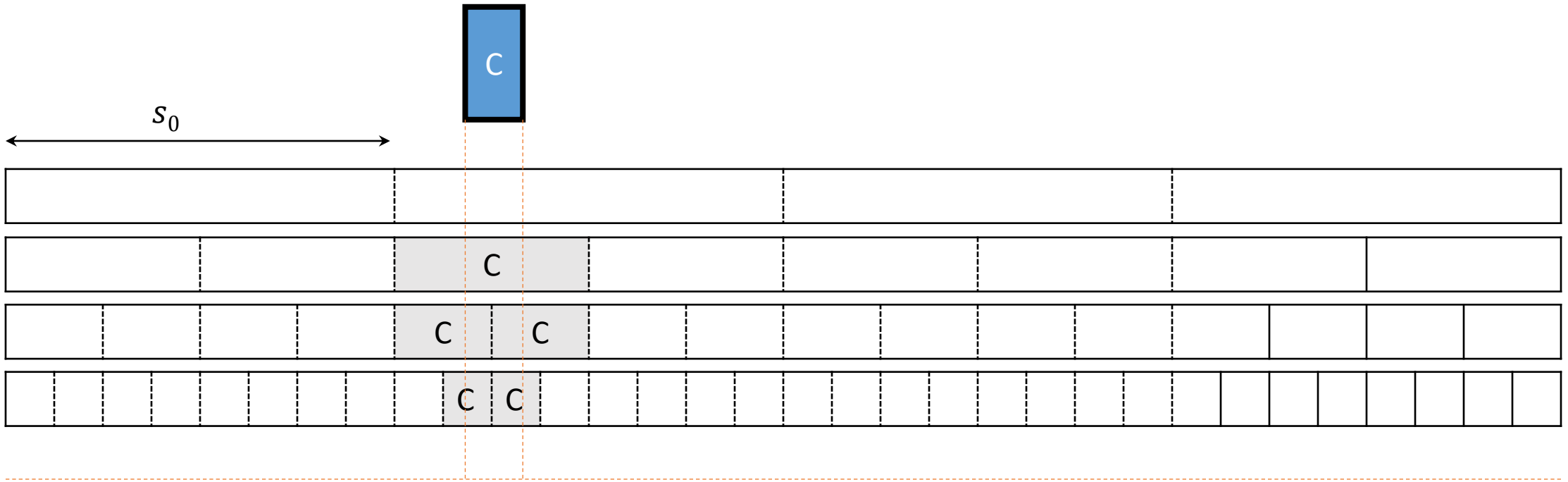
Hierarchical Uniform Grid

- Insert “C” into grid-2



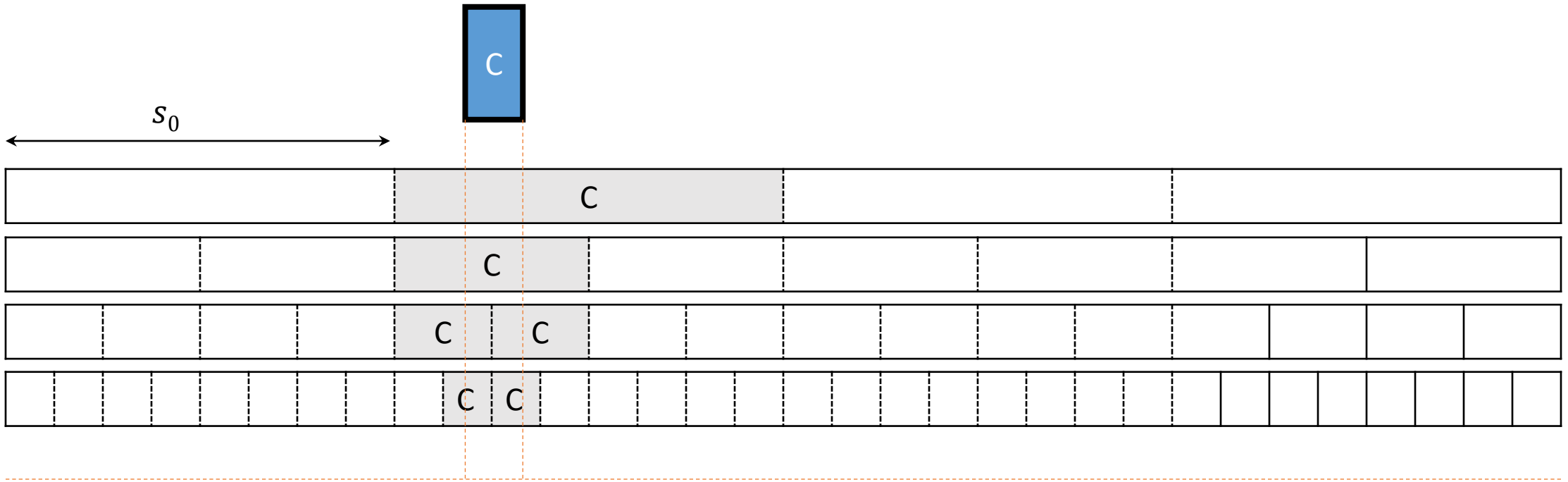
Hierarchical Uniform Grid

- Insert “C” into grid-1



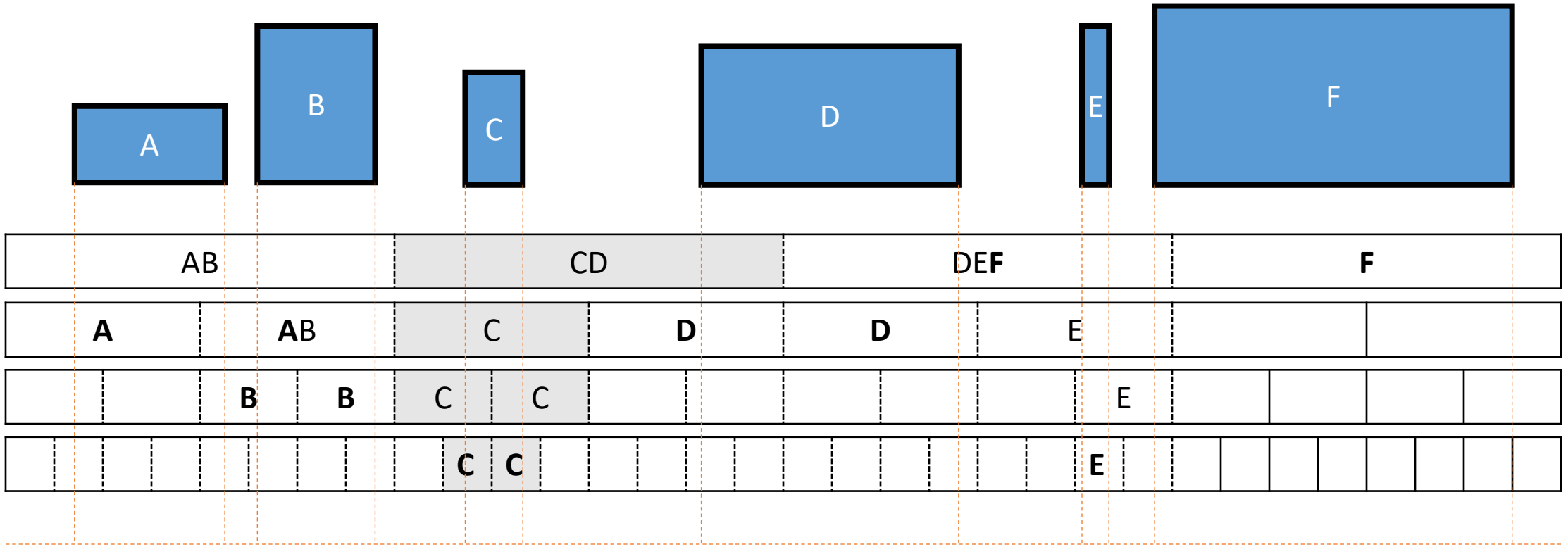
Hierarchical Uniform Grid

- Insert “C” into grid-0



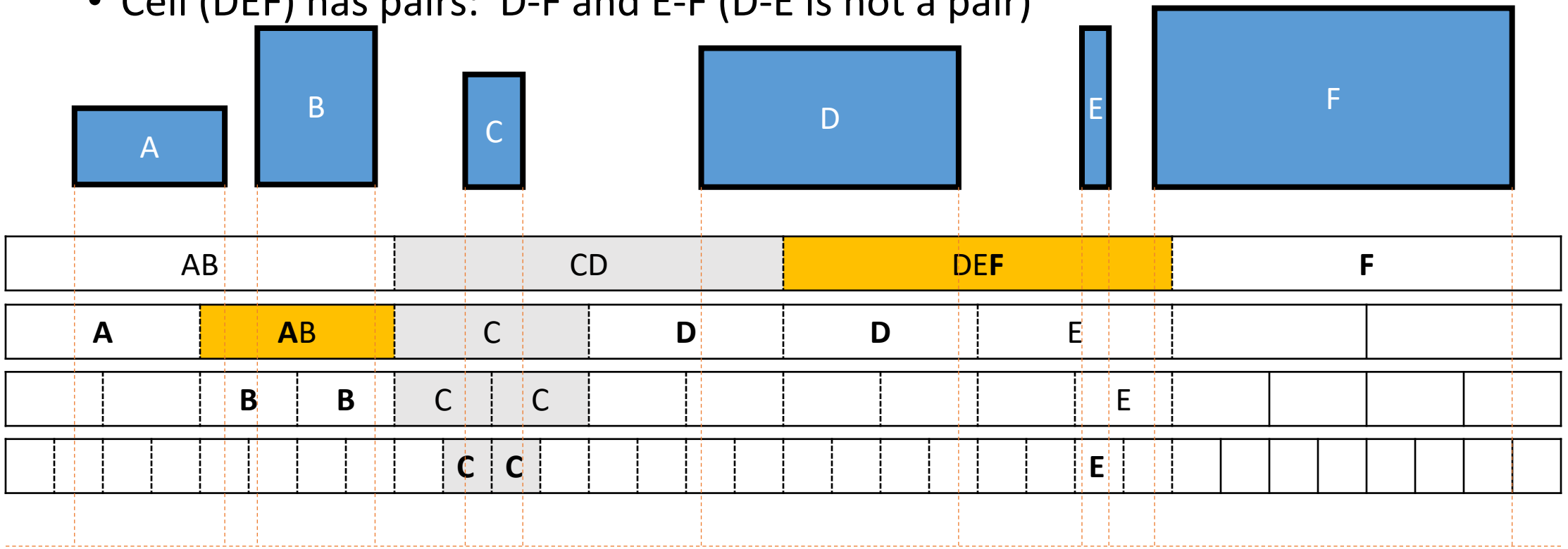
Hierarchical Uniform Grid

- Insert other objects into grids
- Mark IDs to represent the resolution of the object



Hierarchical Uniform Grid

- Report all ID pairs that contain multiple IDs but at least one marked ID
  - Cell (AB) has one pair A-B
  - Cell (DEF) has pairs: D-F and E-F (D-E is not a pair)



Hierarchical Uniform Grid - Methods

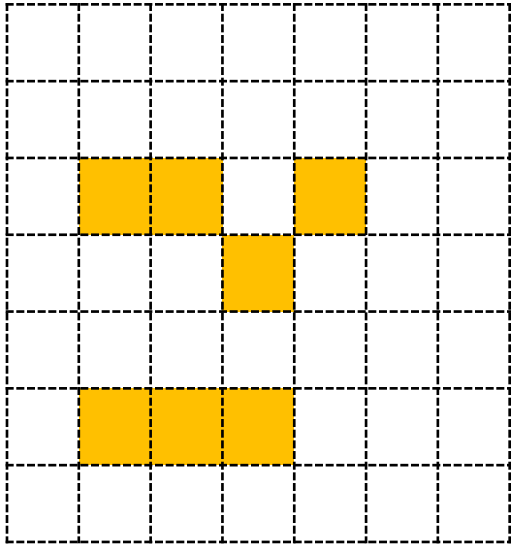
- Add Box
  - Calculate AABB(A), resolution r = Res(A), add box into all grids (0 to r), report pair (A-A_k) only if grid resolution is Min(Res(A), Res(A_k))
- Remove Box
  - Calculate AABB(A), resolution r = Res(A), remove box from all grids (0 to r), remove pairs (A-A_k)
- Update Box
  - Remove Box than Add Box every modified object

Hierarchical Uniform Grid - Summary

- Pros
  - Handle small and large dynamic objects
  - True linear time broad phase algorithm
- Cons
  - More memory (usually 2 times more)
  - Must update more grids for each object
- Constant Update → Linear time complexity
  - Assuming $R = (s^+ / s^-)$ = largest / smallest AABB size is constant
  - We need $k = \log(R)$ grids – constant time
  - One object marks $O(\log R)$ cells – constant time
  - Add/Remove/Update - are constant → time complexity is $O(n)$

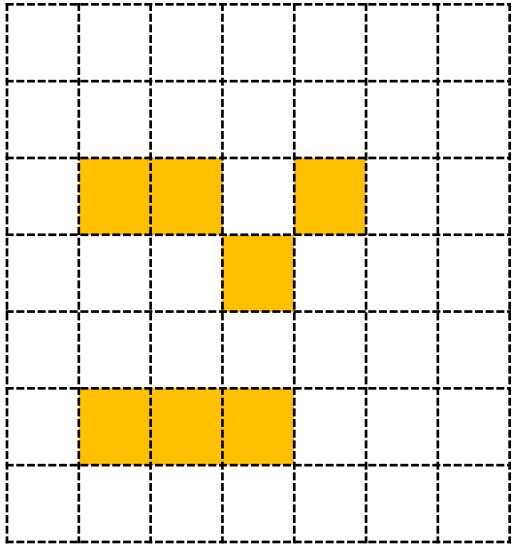
Further Grid Optimization

- Consider the following uniform grid



Further Grid Optimization

- Consider the following uniform grid



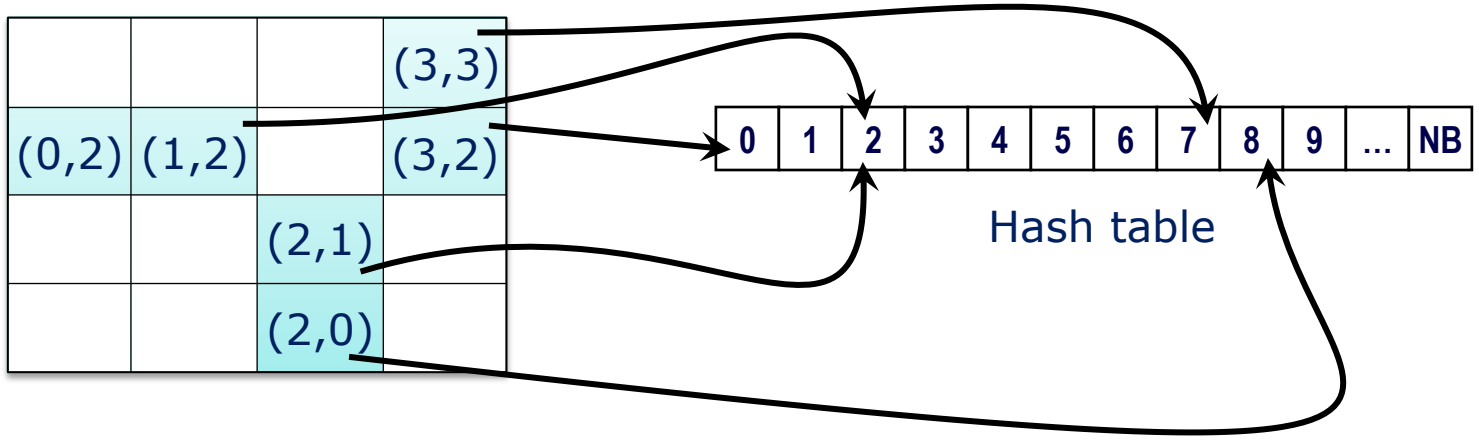
- Most of the cells are empty!

Spatial Hashing

- Motivation: large grids are usually very sparse – we need to store data only for non-empty cells – but we need fast O(1) access based on (x,y,z)

Spatial Hashing

- Motivation: large grids are usually very sparse – we need to store data only for non-empty cells – but we need fast O(1) access based on (x,y,z)



Spatial Hashing

- Motivation: large grids are usually very sparse – we need to store data only for non-empty cells – but we need fast O(1) access based on (x,y,z)
- Given point `p=(x,y,z)` laying within `cell c=(i,j,k)` we define a **spatial hashing function** as

$$\texttt{hash(i,j,k)} = (\texttt{i}p_1 \ \textbf{xor}\ \texttt{j}p_2 \ \textbf{xor}\ \texttt{k}p_3\ )\ \textbf{mod}\ \texttt{n}$$

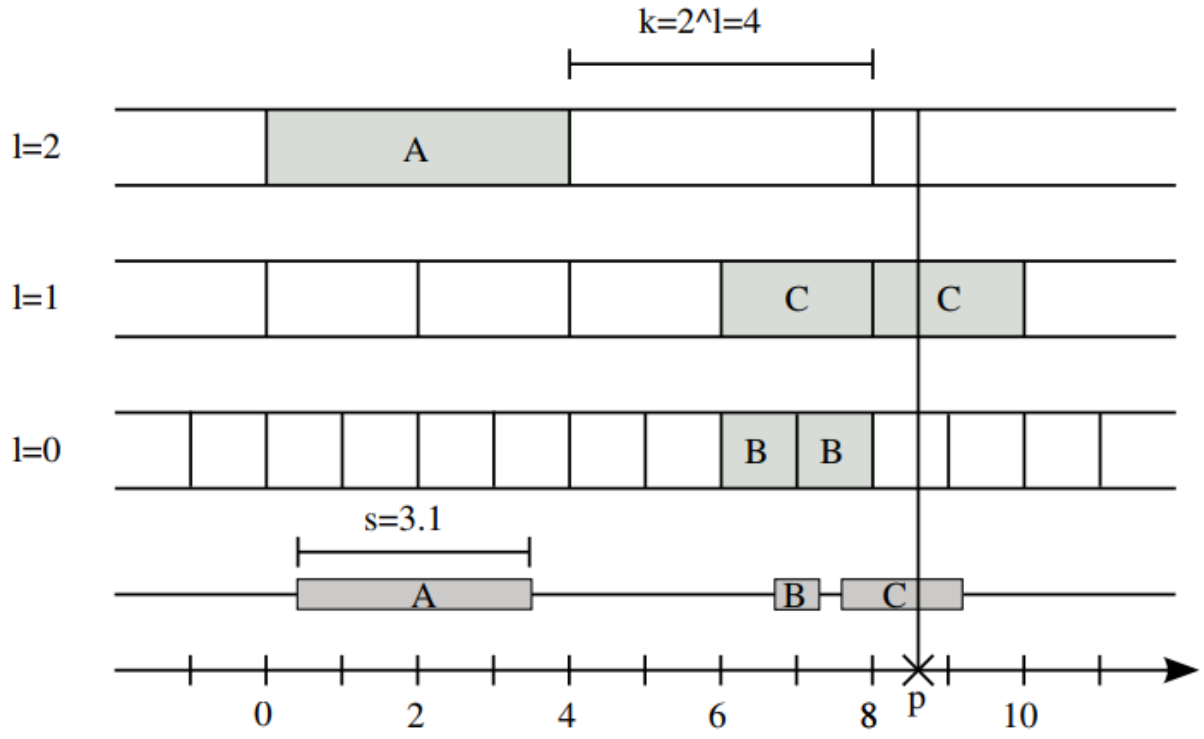
- Where p1 , p2 , p3 are large prime numbers and n is the size of hash table

Matthias Teschner, Bruno Heidelberger, Matthias Muller, Danat Pomeranets, Markus Gross. Optimized Spatial Hashing for Collision Detection of Deformable Objects. 8th Workshop on Vision, Modeling, and Visualization (VMV 2003).

Spatial Hashing: Example

Grid cell size $k = 2^l$ Subdivision level $l = \lceil \log_2(s) \rceil$

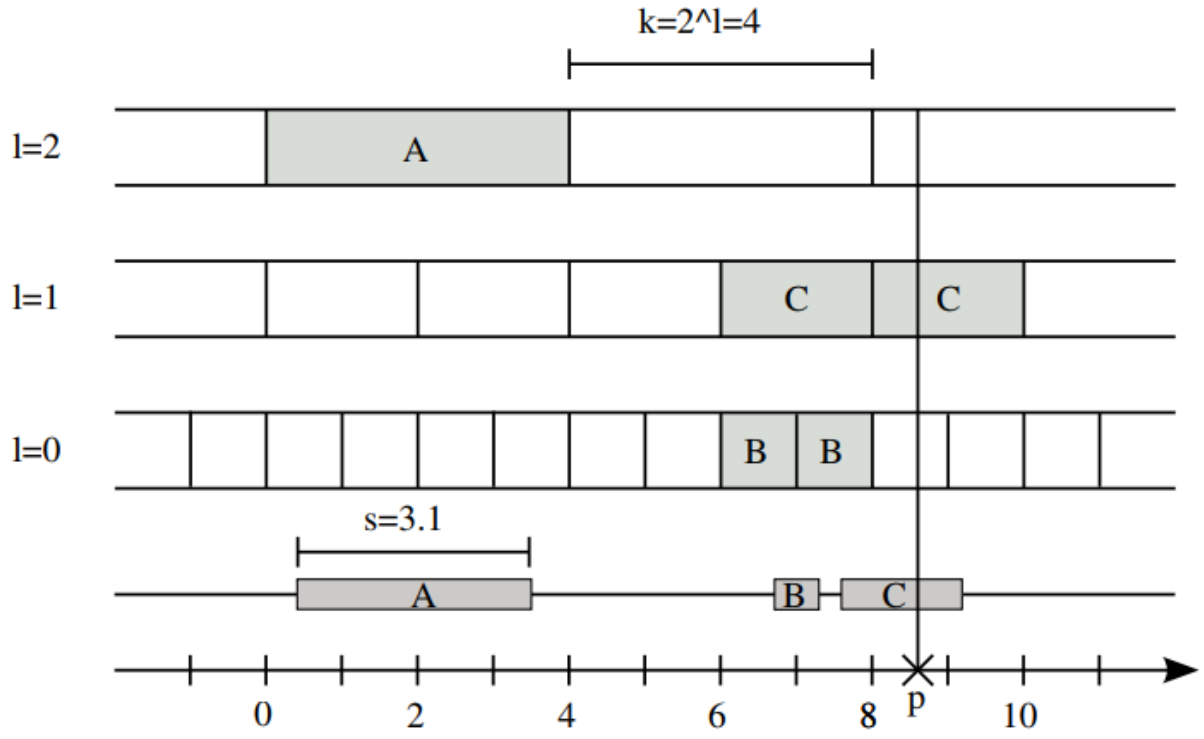
Point → grid address $\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} \lfloor x/k \rfloor \\ \lfloor y/k \rfloor \\ \lfloor z/k \rfloor \\ l \end{bmatrix}$



Spatial Hashing: Example

Grid cell size $k = 2^l$ Subdivision level $l = \lceil \log_2(s) \rceil$

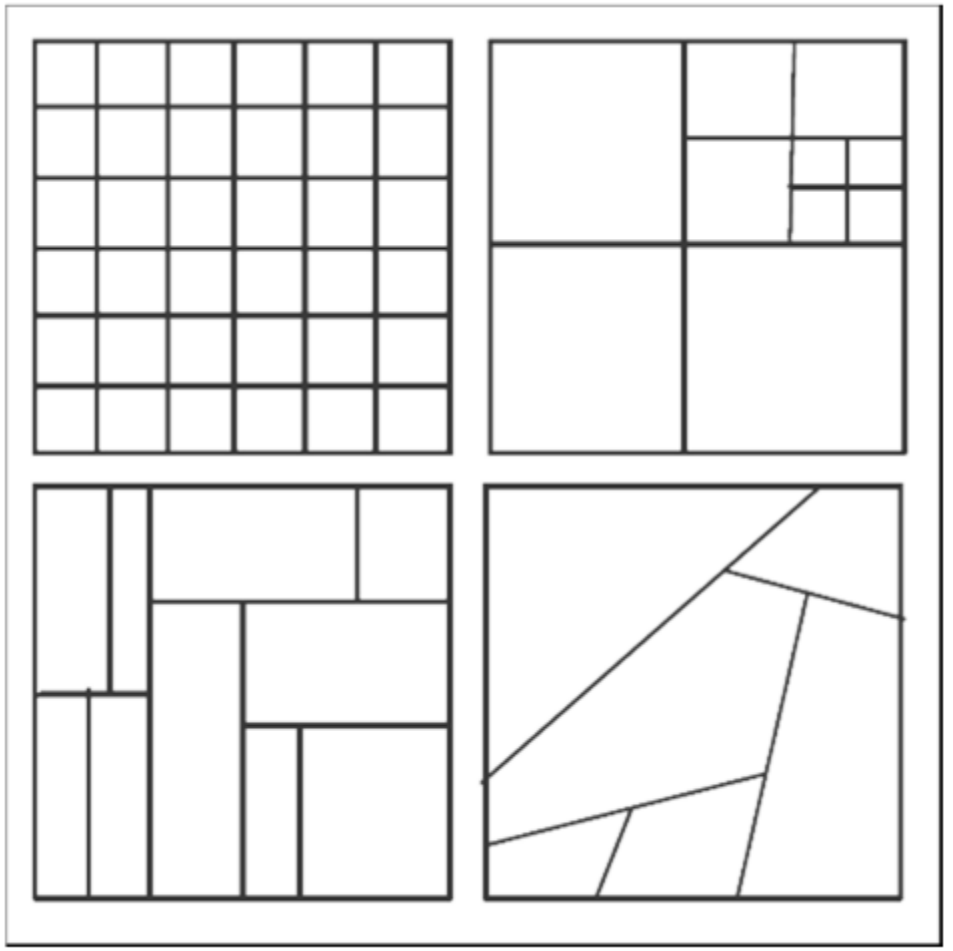
Point → grid address $\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} \lfloor x/k \rfloor \\ \lfloor y/k \rfloor \\ \lfloor z/k \rfloor \\ l \end{bmatrix}$



test against p	index	hash table
—	h(6,0)	B
✓	h(8,0)	empty
—	h(0,2)	A
✓	h(3,1)	C
—	h(7,0)	B
✓	h(2,2)	empty
	h(4,1)	C

Spatial Subdivison Schemes

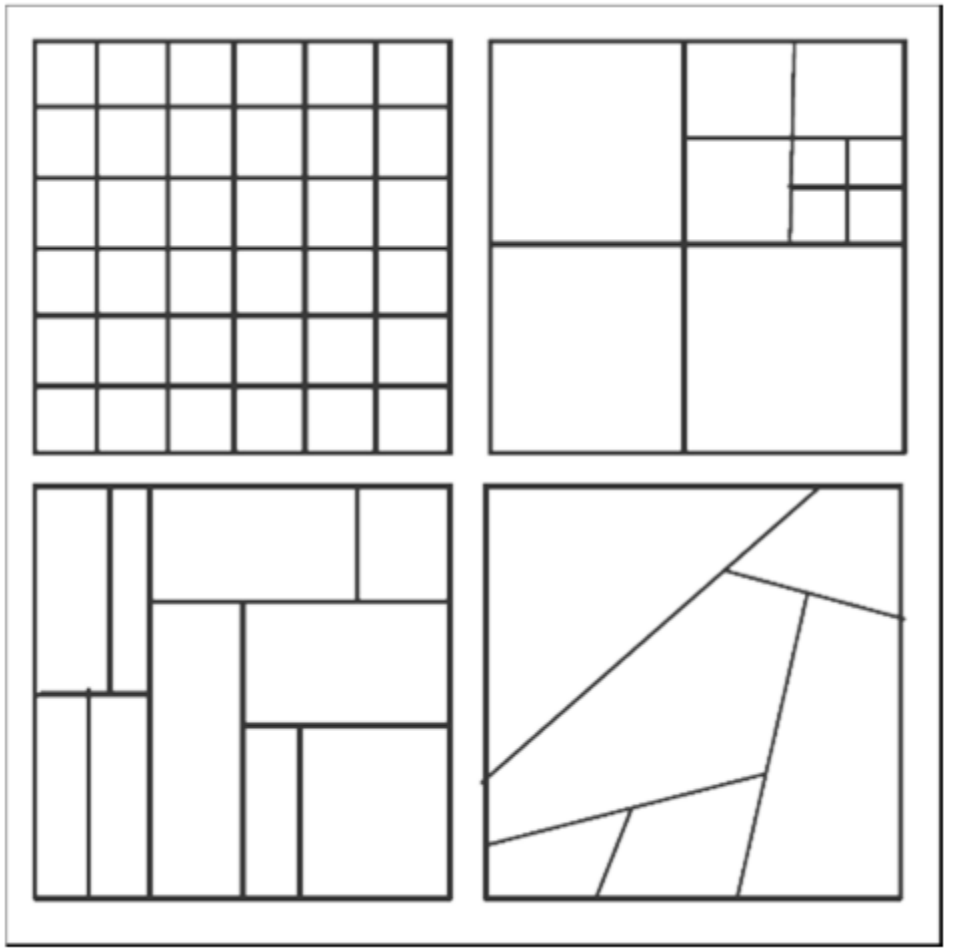
- **Object independent**
  - Grid
- **Object dependent**
  - Quadtree/Octree
  - Kd-tree
  - BSP Tree



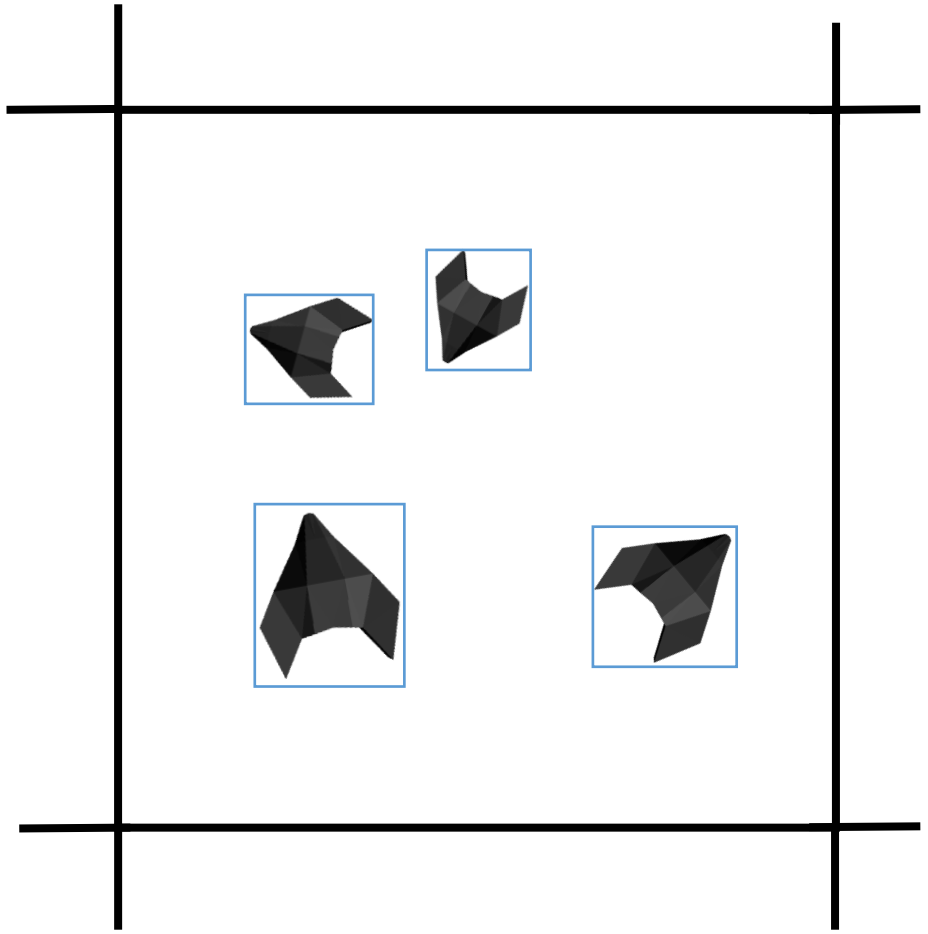
Spatial Subdivison Schemes

- **Object independent**
  - Grid
- **Object dependent**
  - Quadtree/Octree
  - Kd-tree
  - BSP Tree

- Object dependent → not suitable for dynamic scenes



Collisions inside a grid cell?

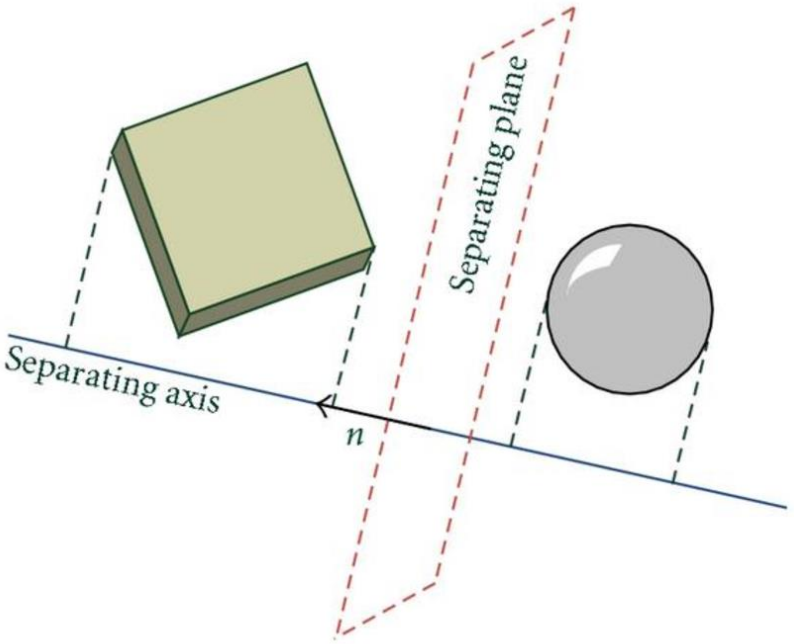


Sweep-And-Prune (SAP)

- Broad phase collision detection algorithm based on Separating Axes Theorem.
- Pros
  - Suitable for physically based motions
  - Exploits spatial and temporal coherence
  - Practical average O(n) broad phase algorithm
- Cons
  - Uses bad fitting axis-aligned boxes (AABB).
  - Not efficient for complex scenes with far away objects
  - Too many collisions for high-velocity objects

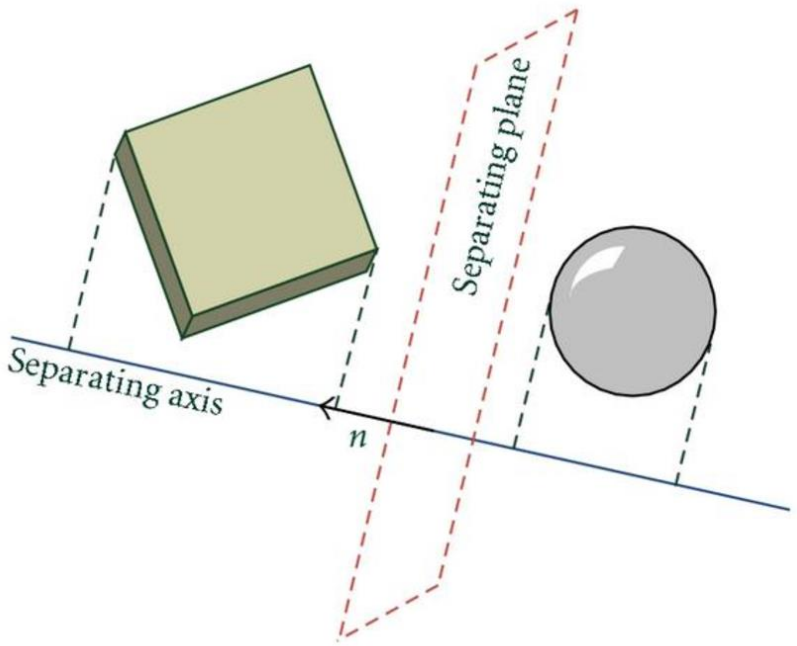
Separating Plane Theorem

- Two convex objects do NOT penetrate (are separated) if and only if there exists a (separating) plane which separates them; i.e. objects are on opposite sides of this plane.



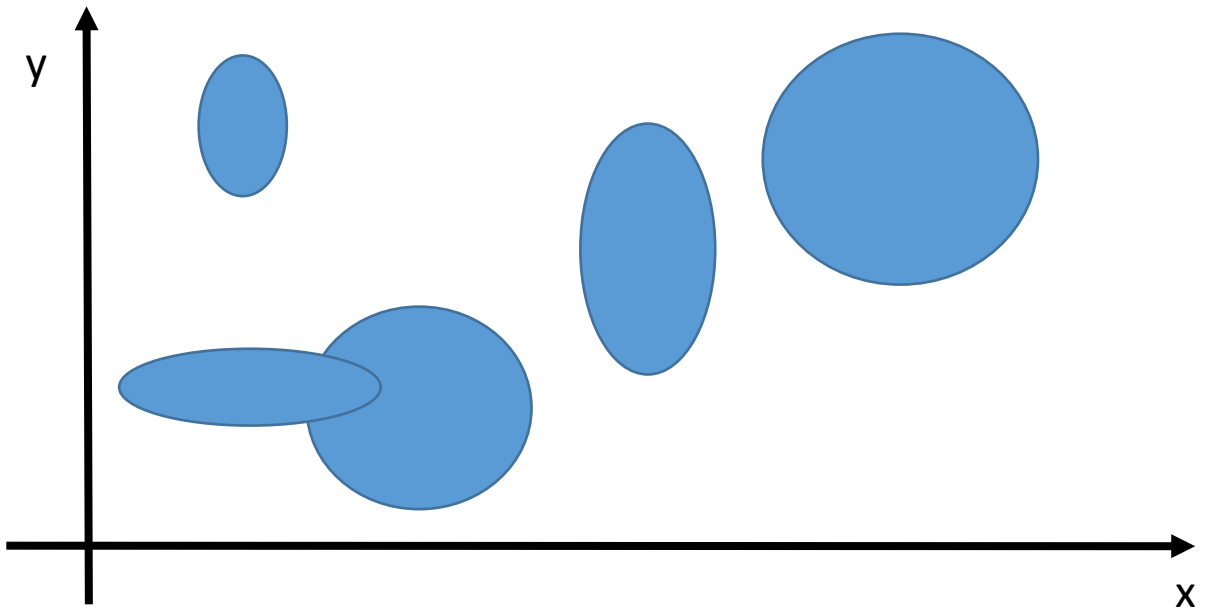
Separating Axis Theorem

- Two convex objects do NOT penetrate (are separated) if and only if there exists a (separating) axis on which projections of objects are separated; i.e. Intervals formed by minimal and maximal projections of objects do not intersect.



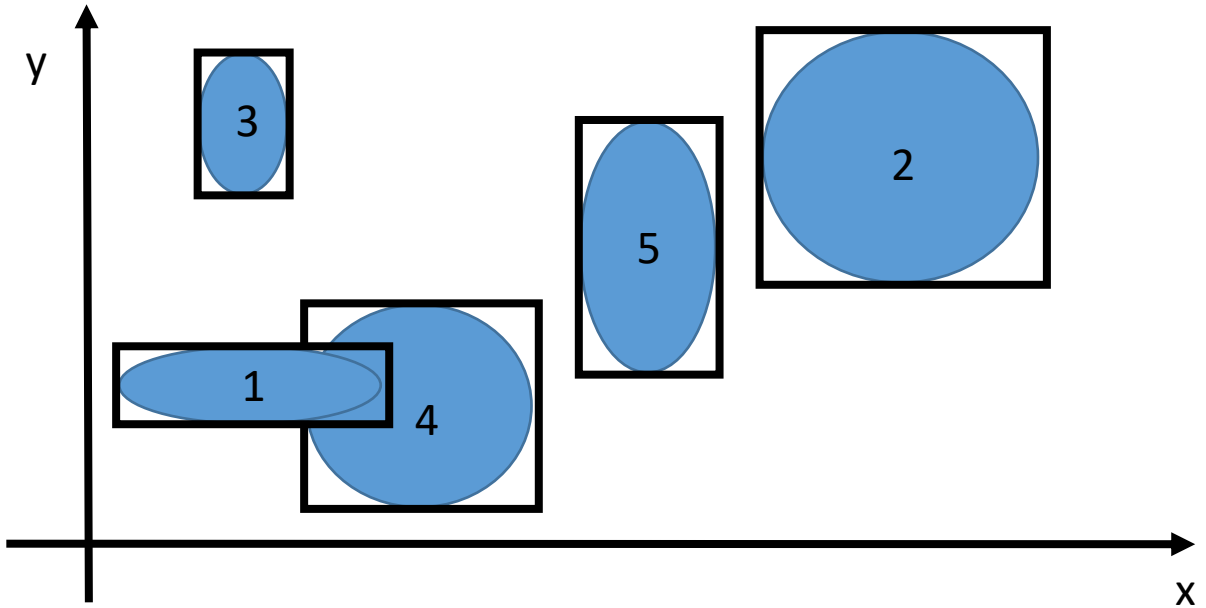
SAP – Algorithm Principle

- Consider a scene with 5 objects



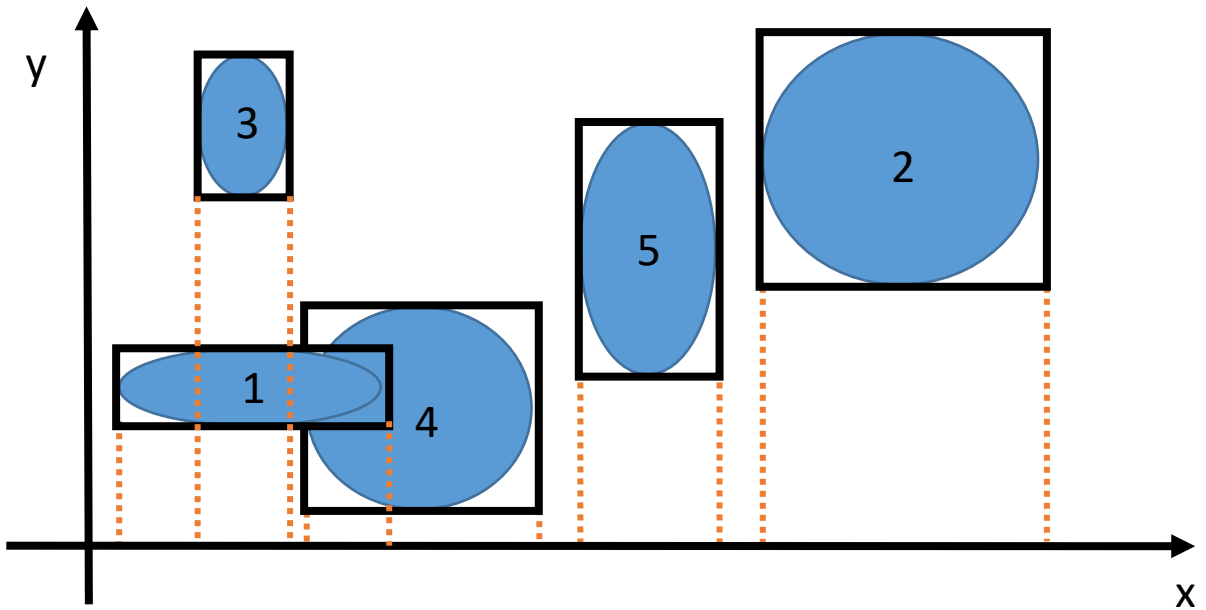
SAP – Algorithm

- Fit each object into its smallest enclosing AABB
- Label boxes as : 1, 2, 3, 4, 5 according to the associated objects.



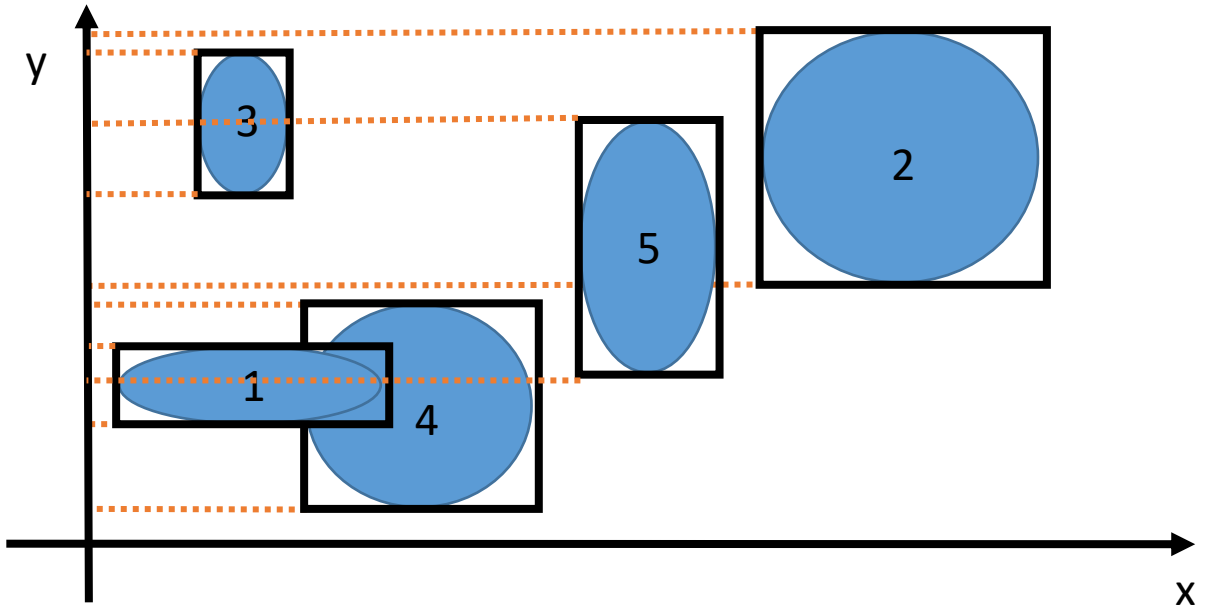
SAP – Algorithm

- Project AABBs onto axis X.
- Form list of intervals of minimal and maximal projections on X axis.



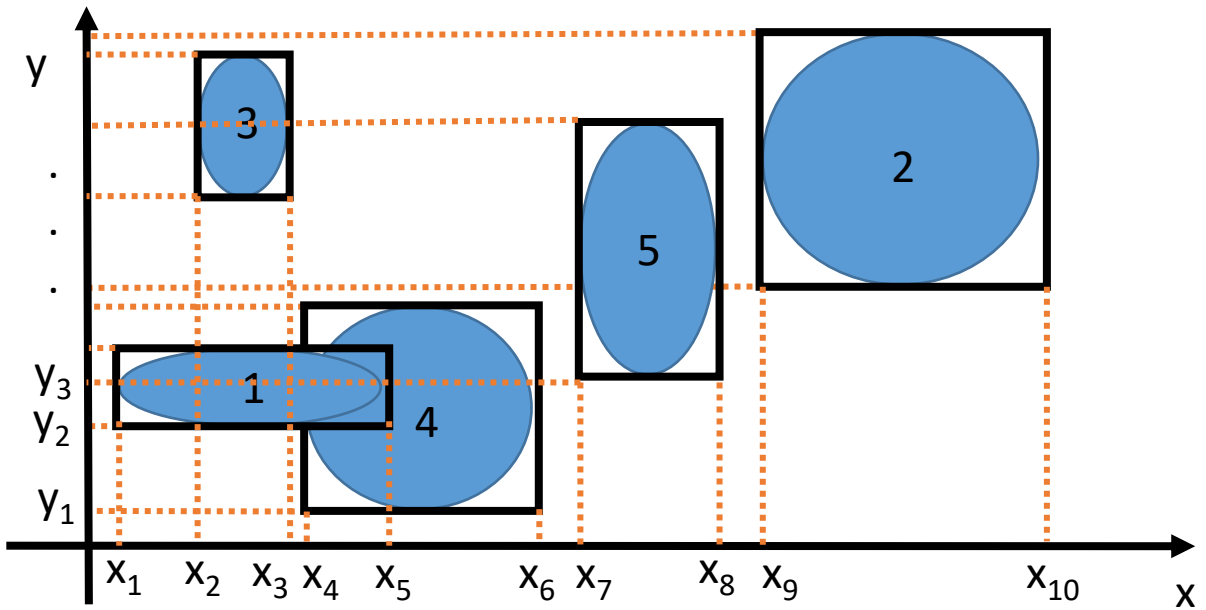
SAP – Algorithm

- Project AABBs onto all other axes.



SAP – Algorithm

- Sort list of projections (limits) on X axis.
- Sort list of projections (limits) on Y axis.

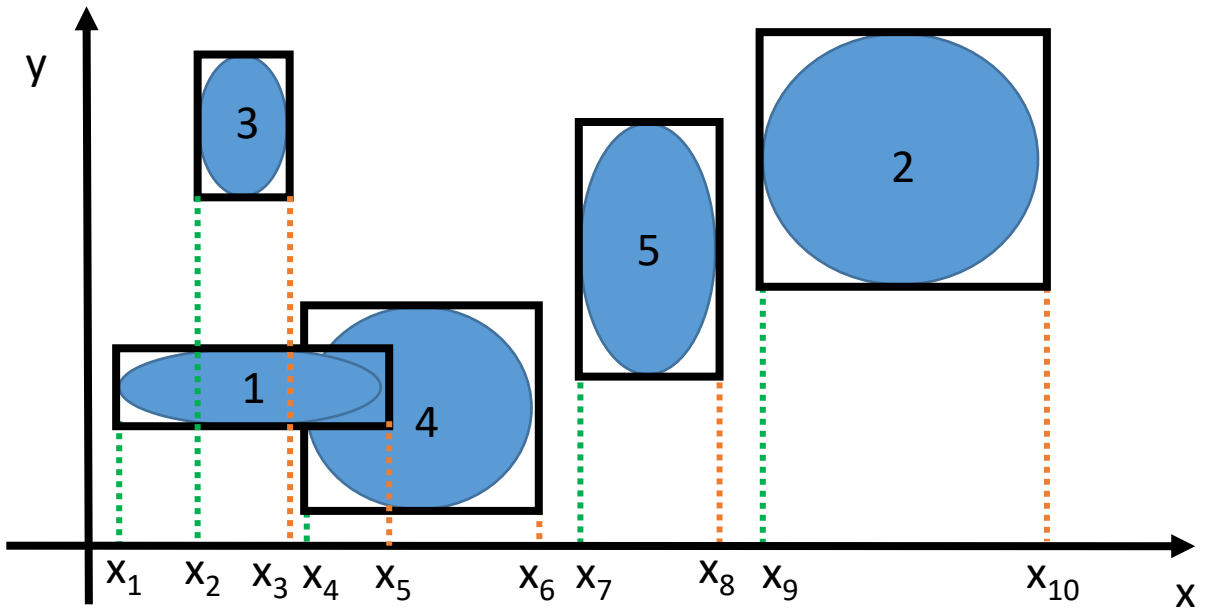


SAP – Algorithm

- Run sweep algorithm
  - Active intervals = ∅;
  - When a min limit value is encountered, add it as intersecting all active intervals; add it to active intervals
  - When max limit value is encountered, remove it from the active intervals
- Intersections must be confirmed across all axes

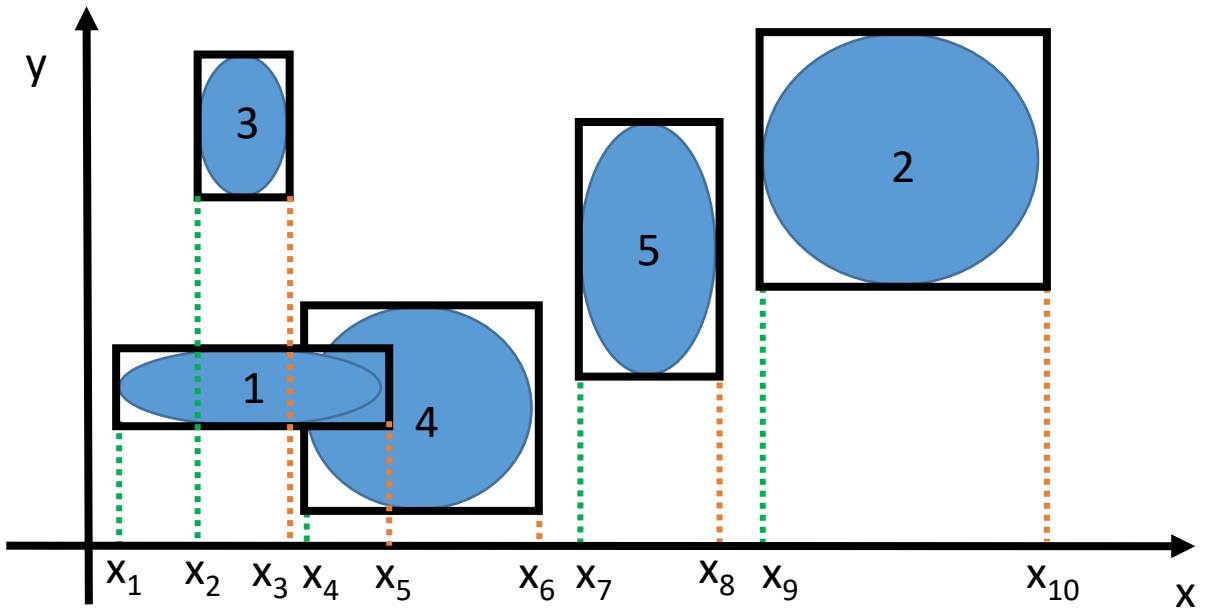
SAP – Algorithm

- Limits are marked as min (green) and max (orange) for associated AABB.



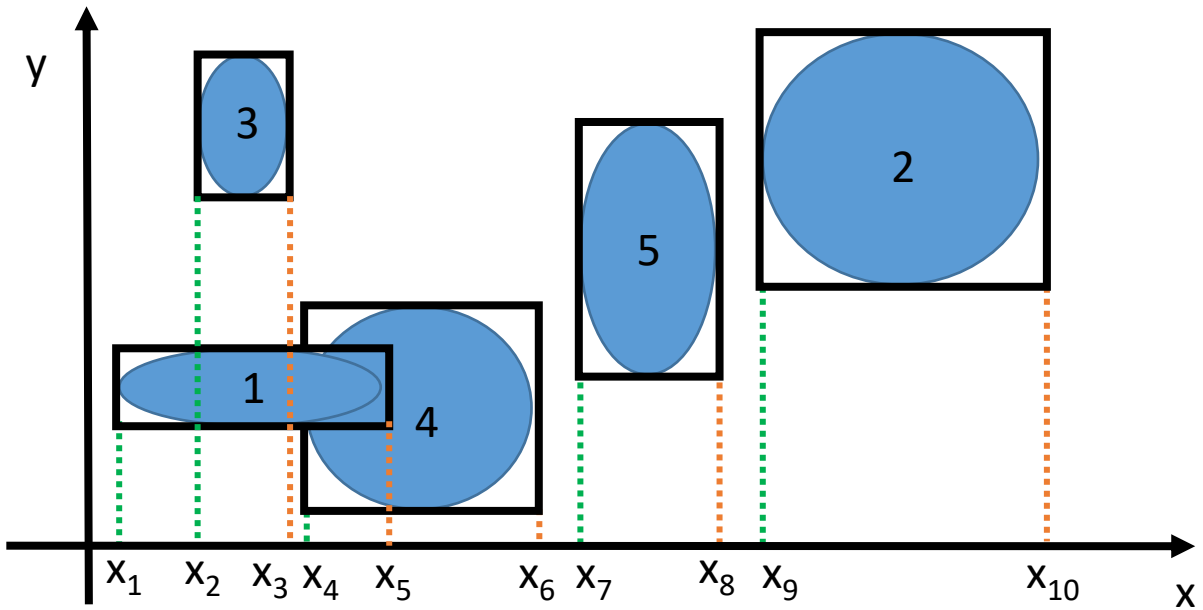
SAP – Algorithm

- Sweep X-limits from first to last while building set of open intervals.
- When adding new min-limit to the set, report potential collision pair between all boxes from set and the new box.



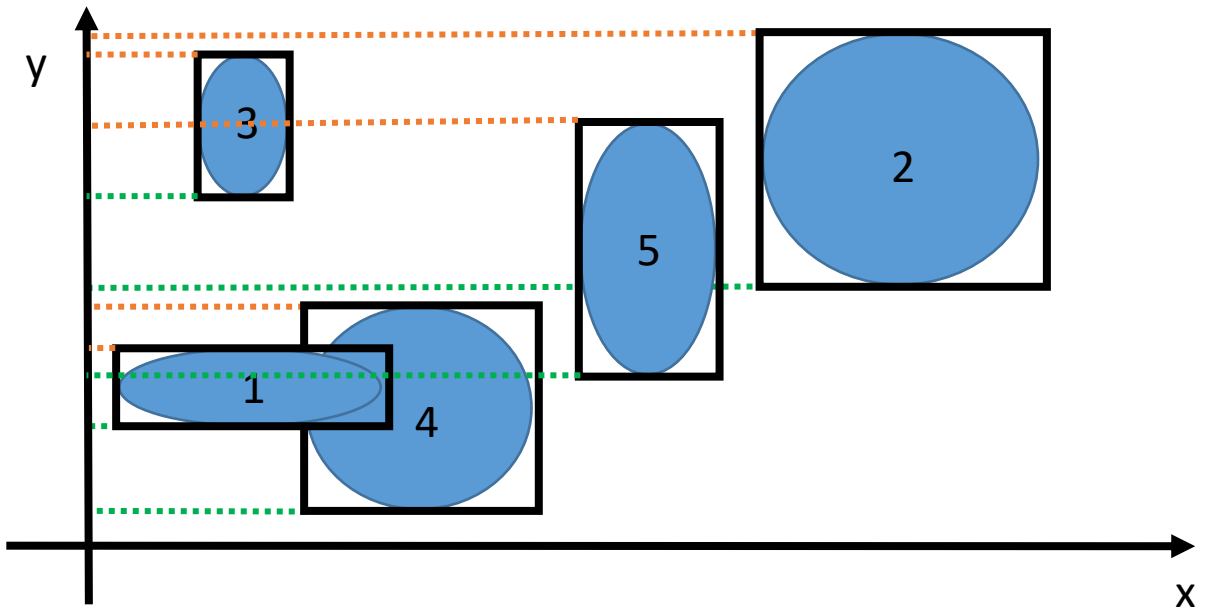
SAP – Algorithm

- Open interval set example:
  - (), (1), (1;3), (1), (1;4), (4), (), (5), (), (2), ()
- Reported pairs: (1-3) and (1-4)



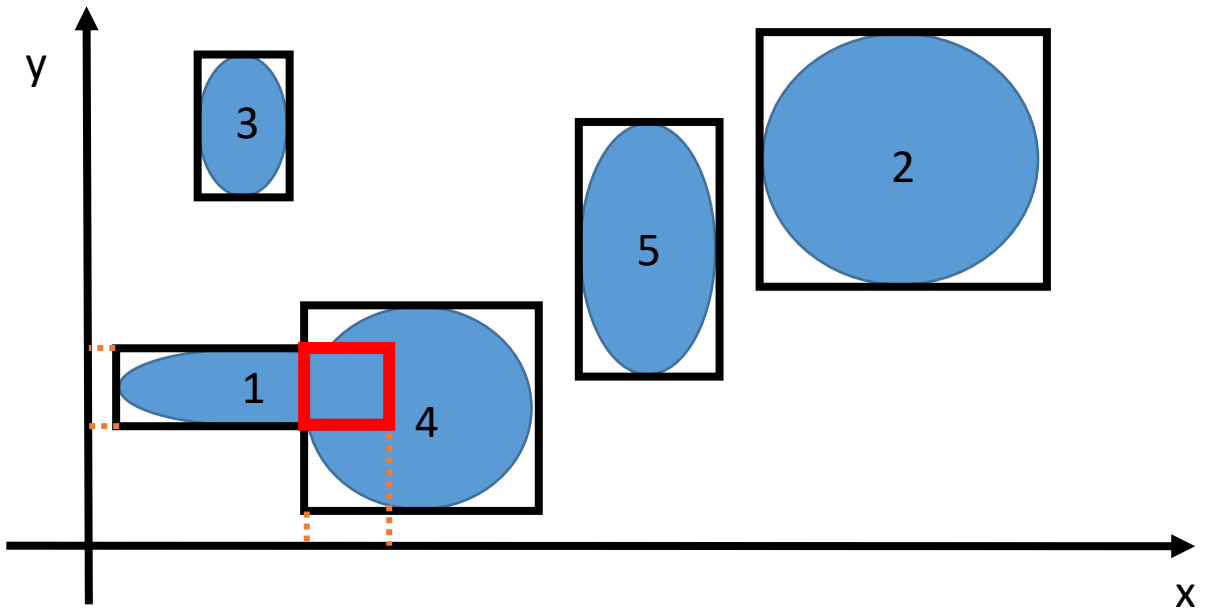
SAP – Algorithm

- Do the same on Y-Axis:
  - Set: (), (4), (4;1), (4;1;5), (4;5), (5), (5;2), (5;2;3), (2;3), (2), ()
  - Pairs: (1-4), (1-5), (4-5), (5-2), (5-3), (2-3)



SAP – Algorithm

- Find common pairs in all swept directions
  - i.e. Real intersecting AABB pairs = SetX ∧ SetY
- Pairs = SetX ∧ SetY = { (1-4) }

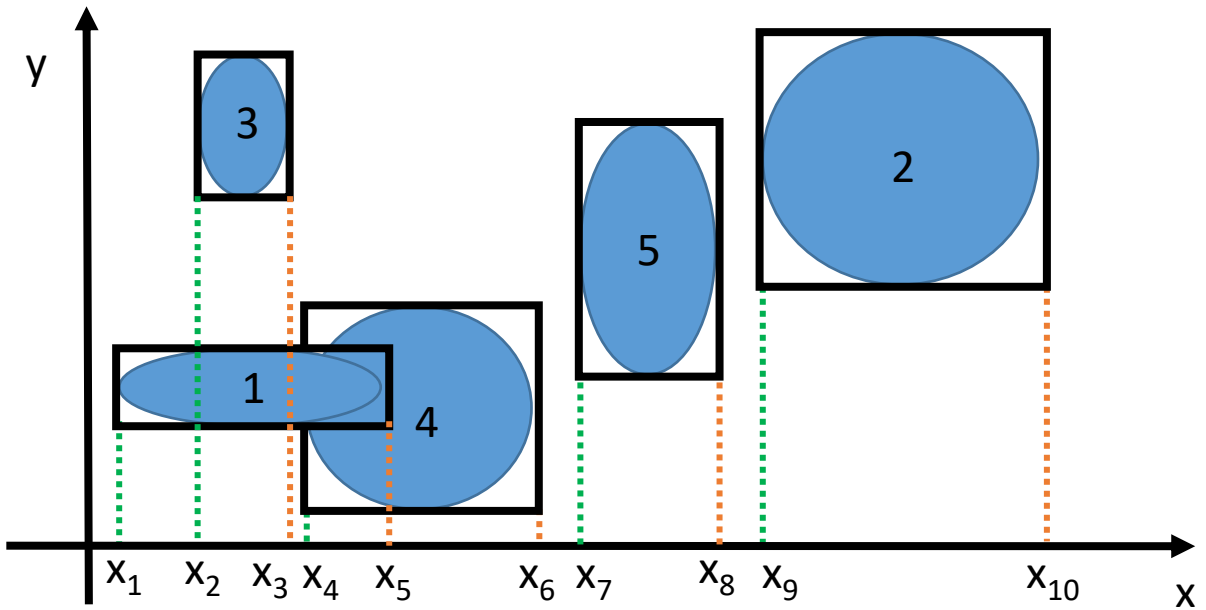


SAP - Summary

- To achieve linear time O(n) complexity in average case we must
  - Move objects in a coherent fashion (physical motion)
  - Use incremental sort of limits. Due to coherence most of limits are sorted. **Insert sort** needs only constant swaps.
  - Initial sort can be done with quicksort.

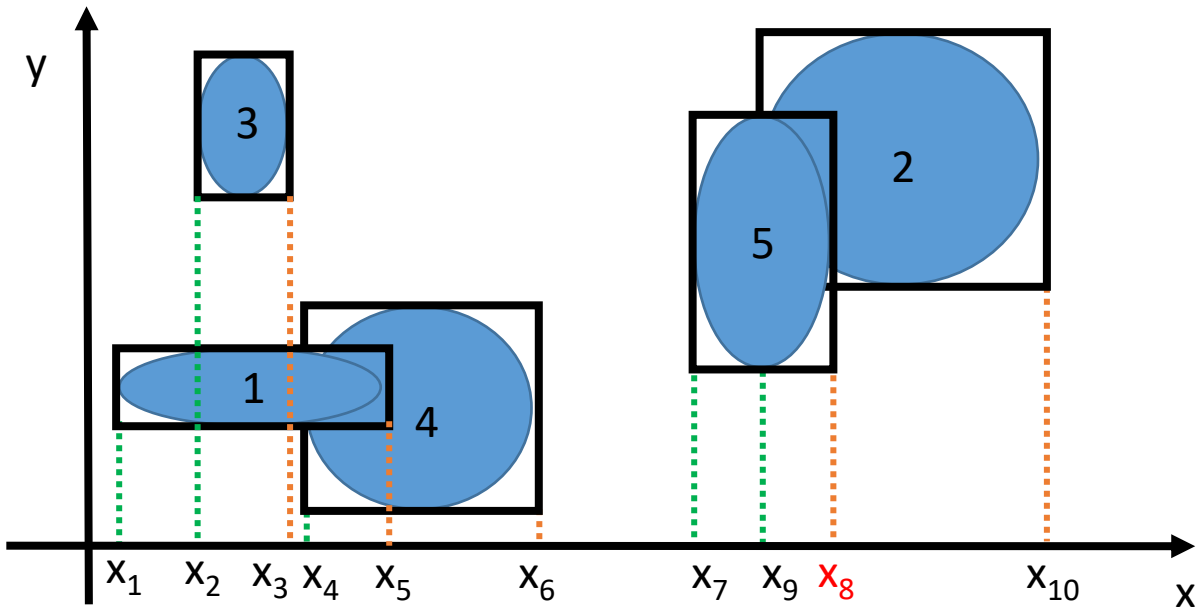
SAP – Incremental Update

- Reported pairs: (1-3) and (1-4)
- Suppose object 5 moves right



SAP – Incremental Update

- End limit x_8 pass over x_9 breaking the order
- In this case we report new pair (2-5)



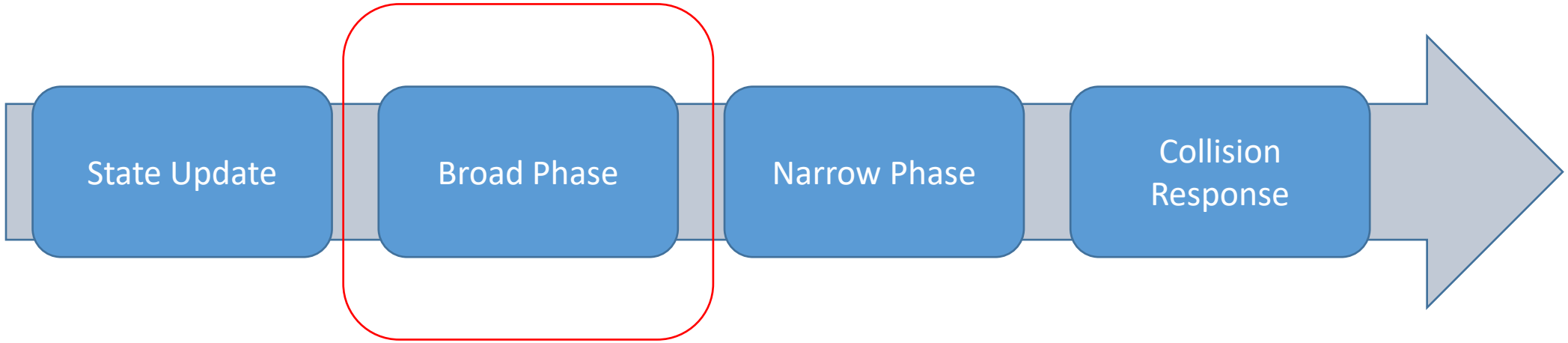
SAP – Incremental Update

- Select moving objects and update their limits
  - When a start limit moves right and
    - passes over start limit – report nothing
    - passes over end limit – remove pair
  - When a start limit moves left and
    - passes over start limit – report nothing
    - passes over end limit – add pair
  - When an end limit moves right and
    - passes over start limit – add pair
    - passes over end limit – report nothing
  - When an end limit moves left and
    - passes over start limit – remove pair
    - passes over end limit – report nothing

Pair Management

- Trivial approach is to use
  - Matrix to store pair infos - just look at (ID1 , ID2) item
  - Simple list to store set of active pairs.
  - Can be efficient for < 1000 objects (matrix size n^2)
- Use spatial hashing to improve efficiency

Where we are at…



Increase Performance of Assignment

- Spatial Subdivision + Spatial Hashing