Broad Phase
Collision Detection

Lesson 04

Lesson 04 Outline

- Collision Detection overview
- Hierarchical grids and Spatial hashing
- Sweep and Prune and Radix Sort
- Pair management – a practical guide
- Demos / tools / libs

Collision Detection Overview

- Collision detection (CD) means
  - Calculate when and where are objects overlapping.
- General taxonomy of algorithms
  - Static / Pseudo-dynamic / Dynamic
- Stages of CD algorithms
  - Broad Phase / (Mid Phase) / Narrow Phase
- Algorithm strategies
  - Spatial partitioning / Bounding volume hierarchies / Coordinate sorting / Feature tracking / Signed distance maps ...

Broad Phase

- Approximate (broad) collision detection phase.
- Principles
  - Quickly find pairs of objects which are potentially (probably) colliding.
  - Reject pairs of objects which are distant to each other.
- Techniques
  - Uniform Spatial partitioning (Hierarchical grids)
  - Complex Spatial partitioning (dynamic BSP, kd trees)
  - Coordinate sorting (Sweep and prune, range search)
- Difficult to parallelize (GPU not friendly)

Mid Phase

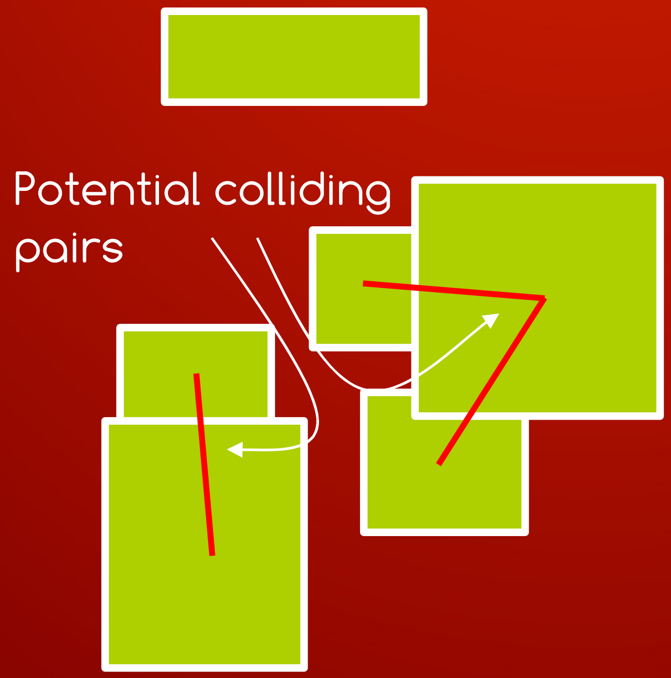
- Mid (refinement) collision detection phase
- Principles
  - Refine pairs from broad phase, simplify the work of narrow phase
- Techniques
  - Preprocess complex geometry into Bounding Volume Hierarchies
  - Decompose non-convex objects into convex parts
  - Axis Aligned Bounding Boxes, Oriented Bounding Boxes, k-Discrete Orientation Polytopes, Swept Sphere Volumes...
- Usually good for parallelization (GPU friendly)

Narrow Phase

- Exact (Narrow) Collision detection phase.
- Principles
  - Given a list of potential colliding pairs of objects find exact time and geometry features (vertices, edges, faces) where objects penetrate (intersect).
  - Reject all non-colliding object pairs.
- Techniques
  - Bounding volume hierarchies (AABB, OBB, kDOP ...)
  - Coherent feature tracking (GJK, V-Clip)
  - Signed distance map queries (2d/3d bitmap collisions)
- Naturally suitable for parallelization (GPU friendly)

Collision Detection Phases

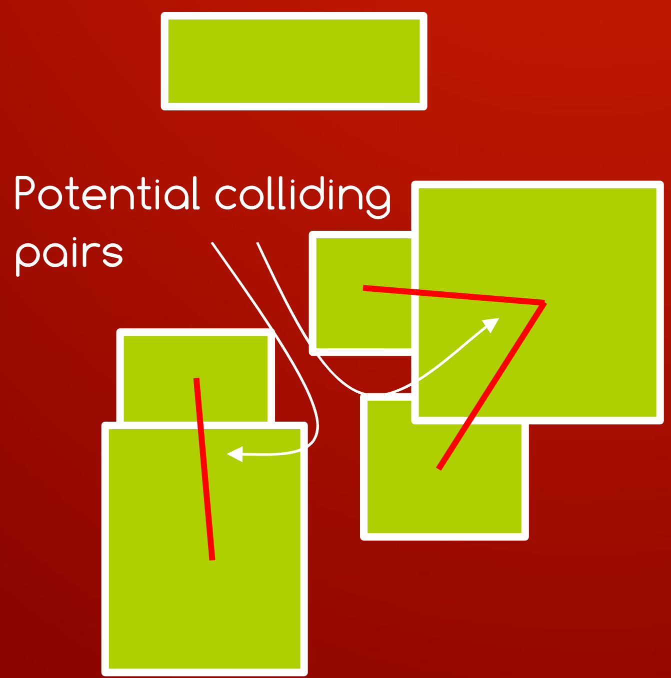
- Broad Phase
- Find potential pairs

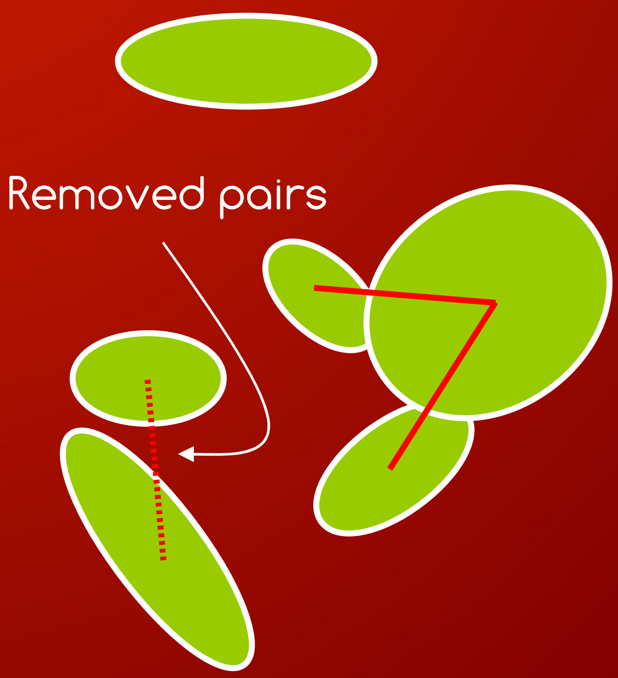


Collision Detection Phases

- Broad Phase
- Find potential pairs

- Mid Phase
- Refine pairs



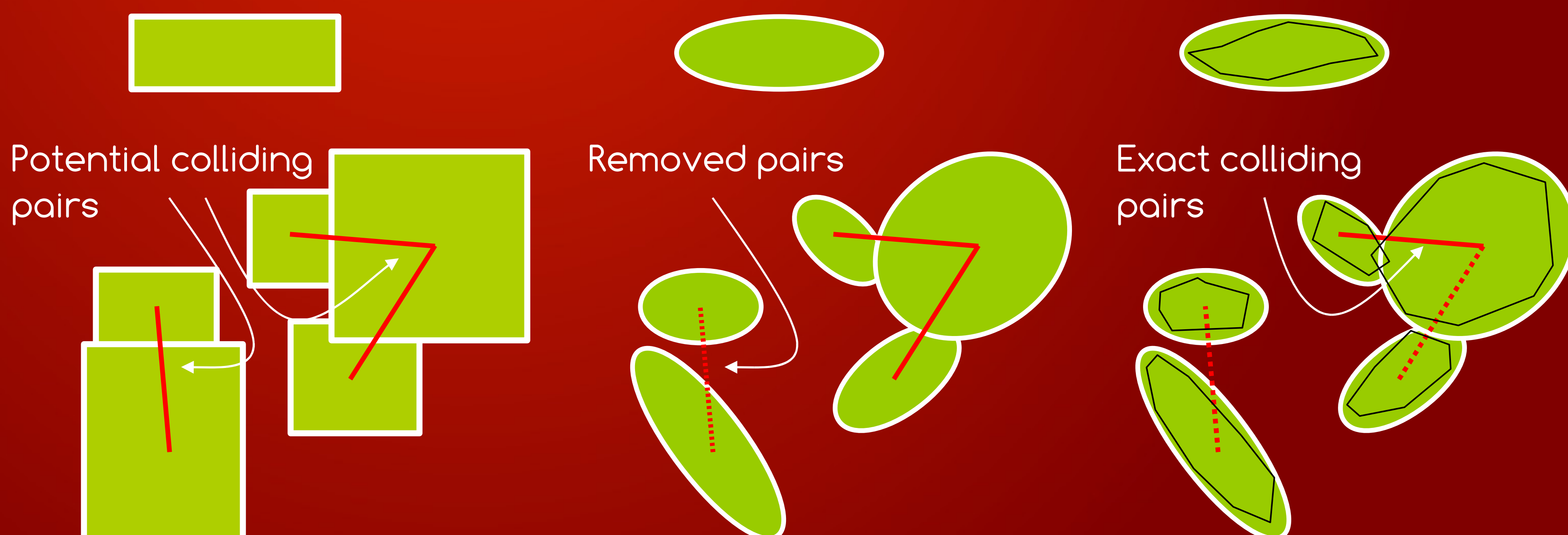


Collision Detection Phases

- Broad Phase
- Find potential pairs

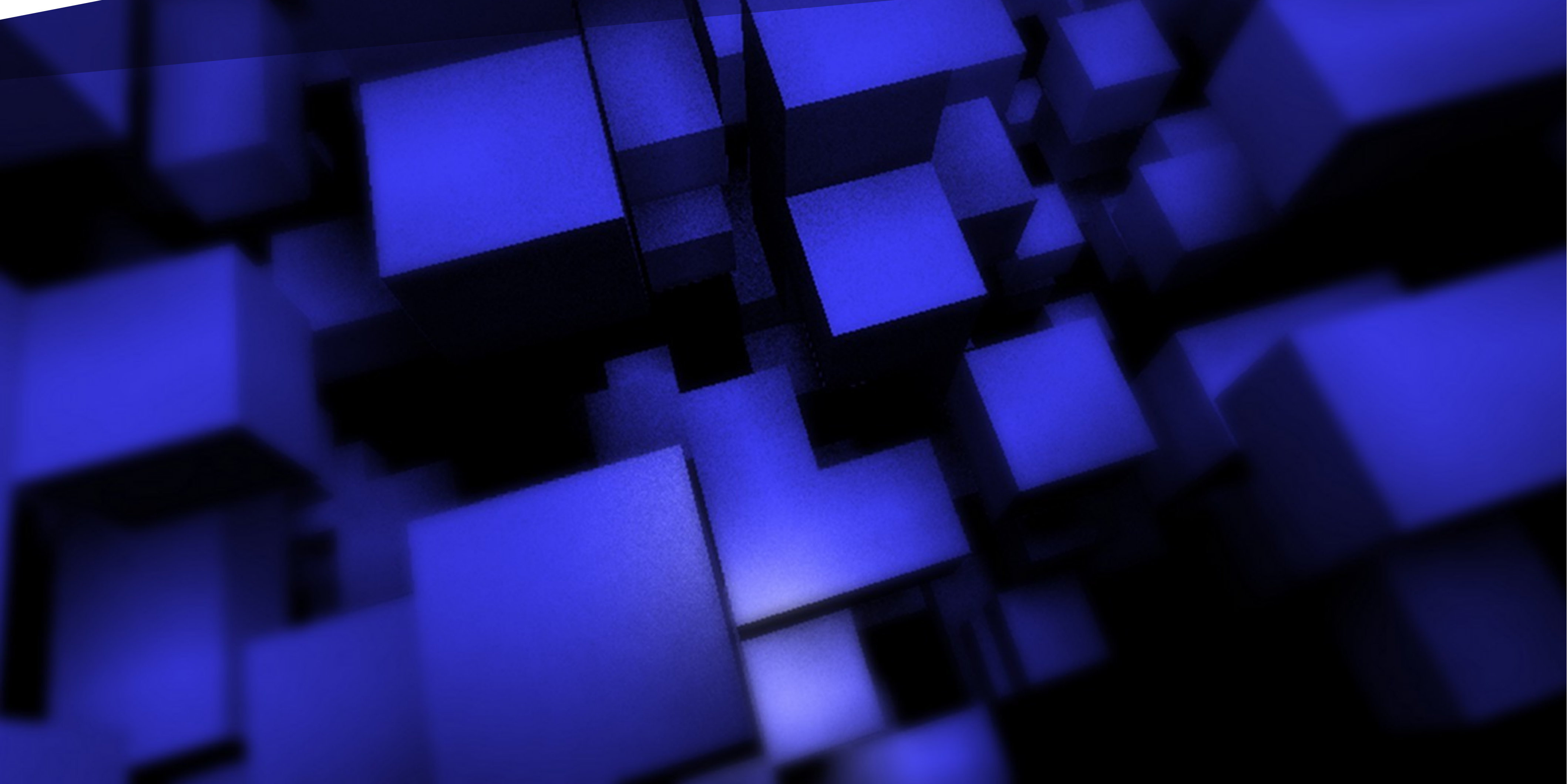
- Mid Phase
- Refine pairs

- Narrow Phase
- Find exact collisions



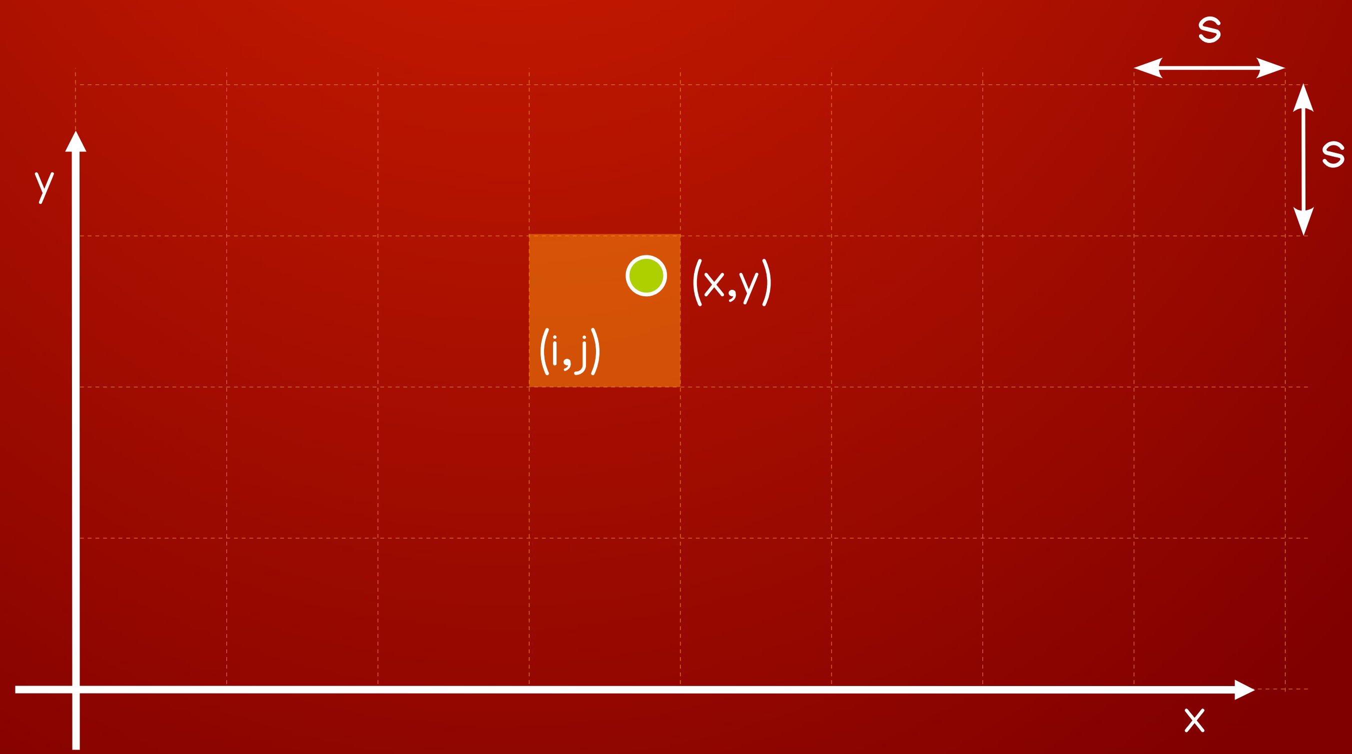
Hierarchical Grids

Spatial Hashing



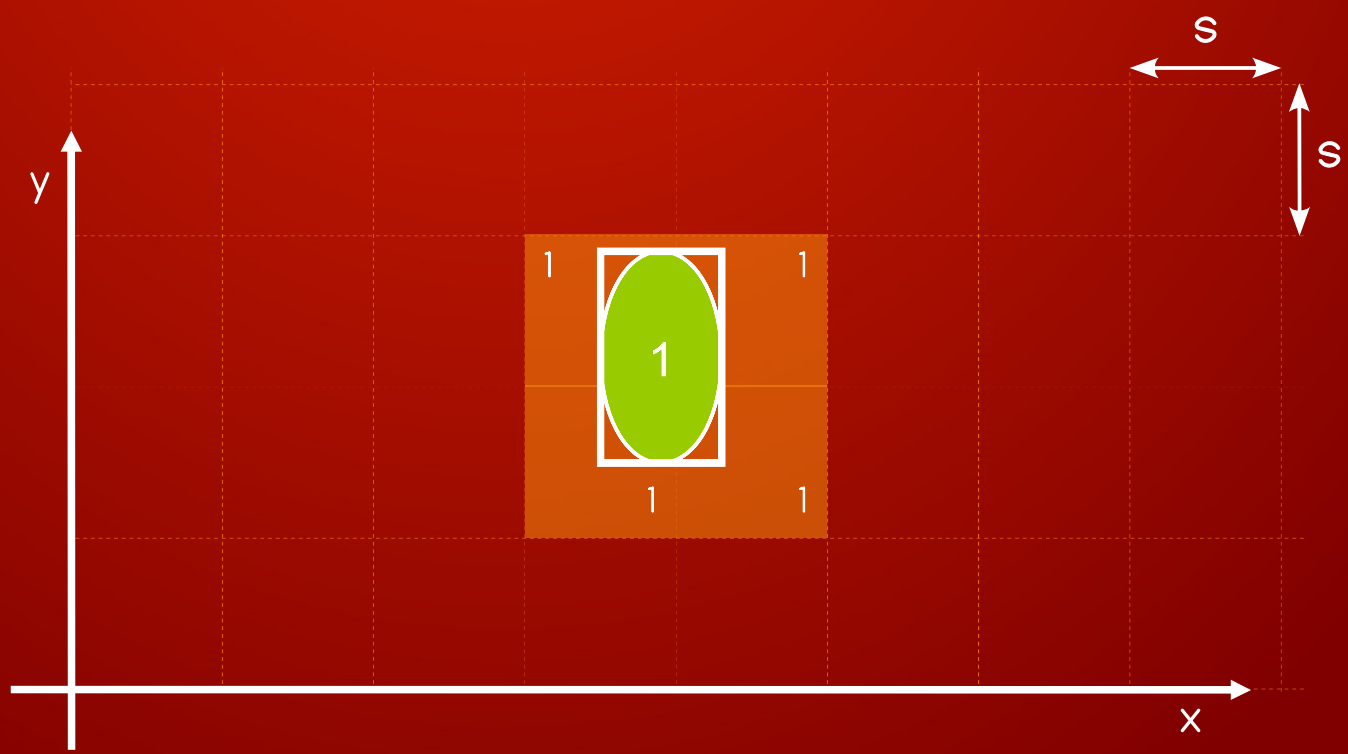
Uniform Grid – Principle

- 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])



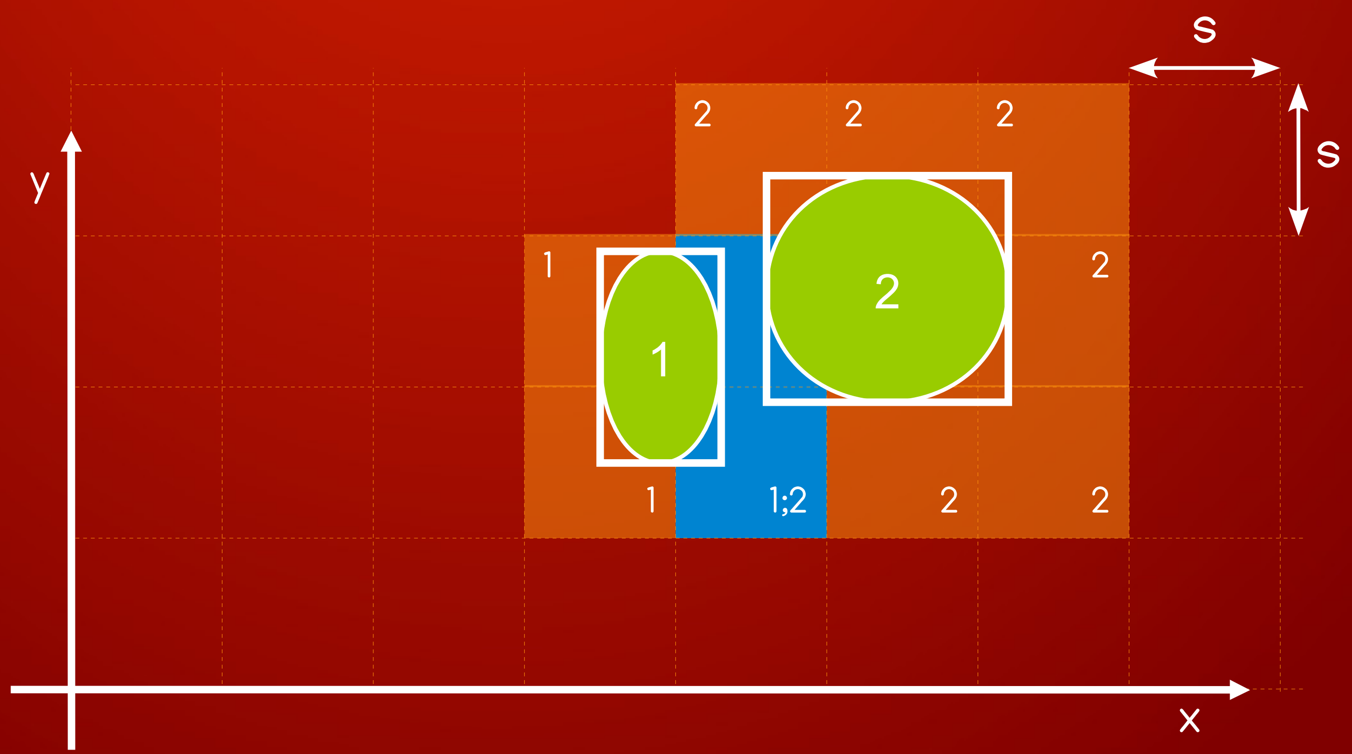
Uniform Grid – Principle

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



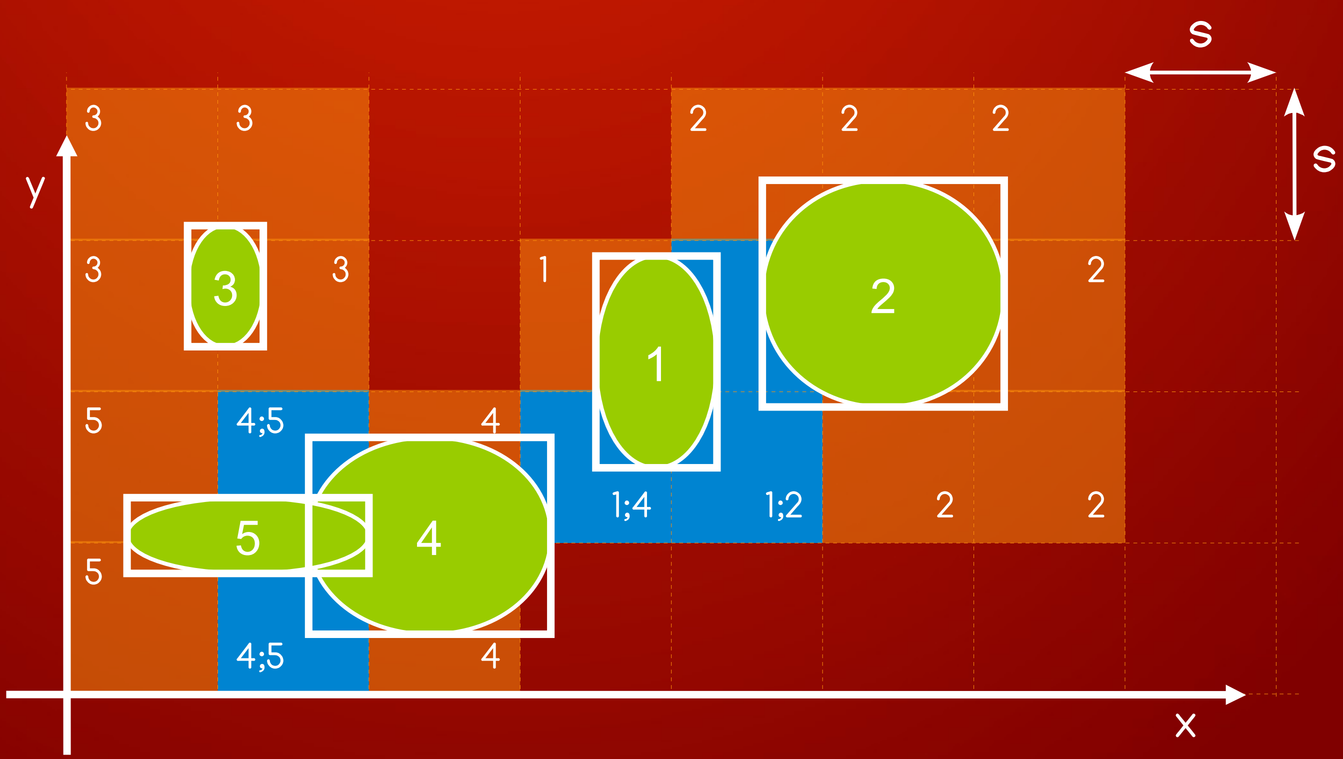
Uniform Grid – Principle

- 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 – Principle

- Insert all objects into grid and store IDs into cells
  - Orange cell has only one ID
  - Blue cells contain more Ids - **define colliding pairs**
- Colliding pairs: (1-2), (1-4), (4-5)



Uniform Grid – AddBox

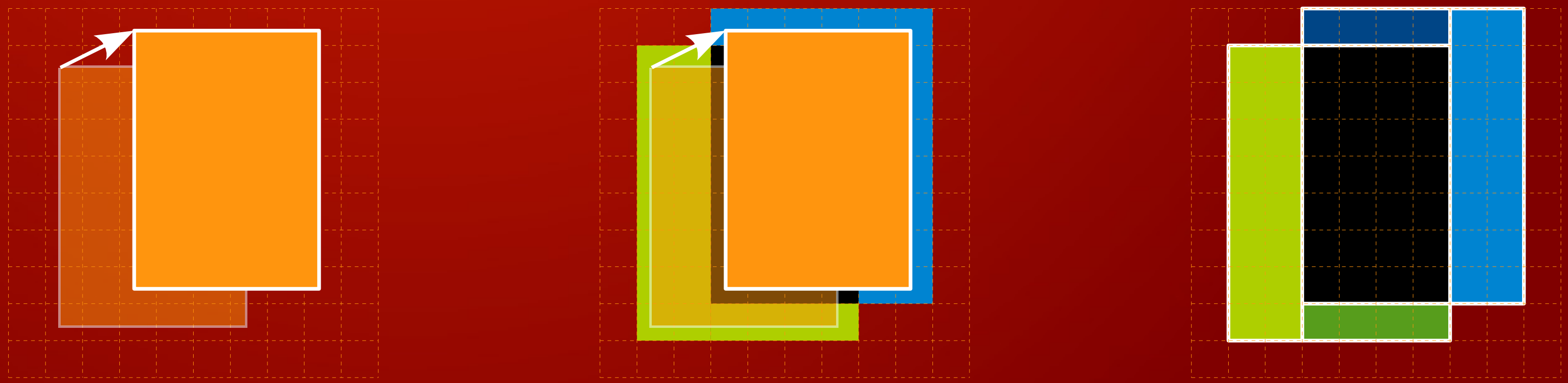
- We want to add new object "A" into grid
- Calculate AABB(A) = $(A_{x-}, A_{y-}, A_{z-}, A_{x+}, A_{y+}, A_{z+})$ of "A"
- Calculate Cell(A) = $(A_{i-}, A_{j-}, A_{k-}, A_{i+}, A_{j+}, A_{k+})$
- For each cell within (A_{i-}, A_{j-}, A_{k-}) and (A_{i+}, A_{j+}, A_{k+})
  - For each ID stored in the cell create pair (ID_k, ID)
  - Add ID of object from the list of IDs (check duplicates)

Uniform Grid – RemoveBox

- We want to remove existing object from grid
- Calculate 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_k, ID)
  - Remove ID of object from the list of IDs

Uniform Grid – UpdateBox

- Object has moved - we need to update it's AABB and corresponding cells
- Simple approach: call RemoveBox, than AddBox
  - Not efficient for larger and coherent objects – many cells has not changed their state (no add, no remove)
- Effective approach:
  - Find quickly only cells where we need to add/remove ID

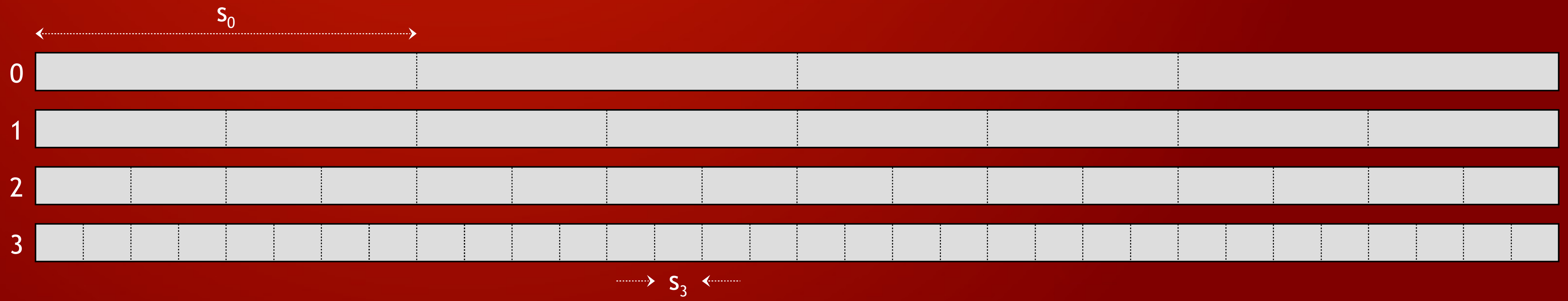


Uniform Grid - Summary

- Pros
  - Simple algorithm – easy to implement
  - Fast in special cases – only particles (small dynamic objects) and static (large) environment
- Cons
  - how to find optimal grid size → problem with large vs small dynamic objects (hierarchical grid)
  - Large 3d gird → huge amount of memory (spatial hashing)
  - Slow grid update for large objects
  - Accuracy depends on the largest resolution

Hierarchical Uniform Grid

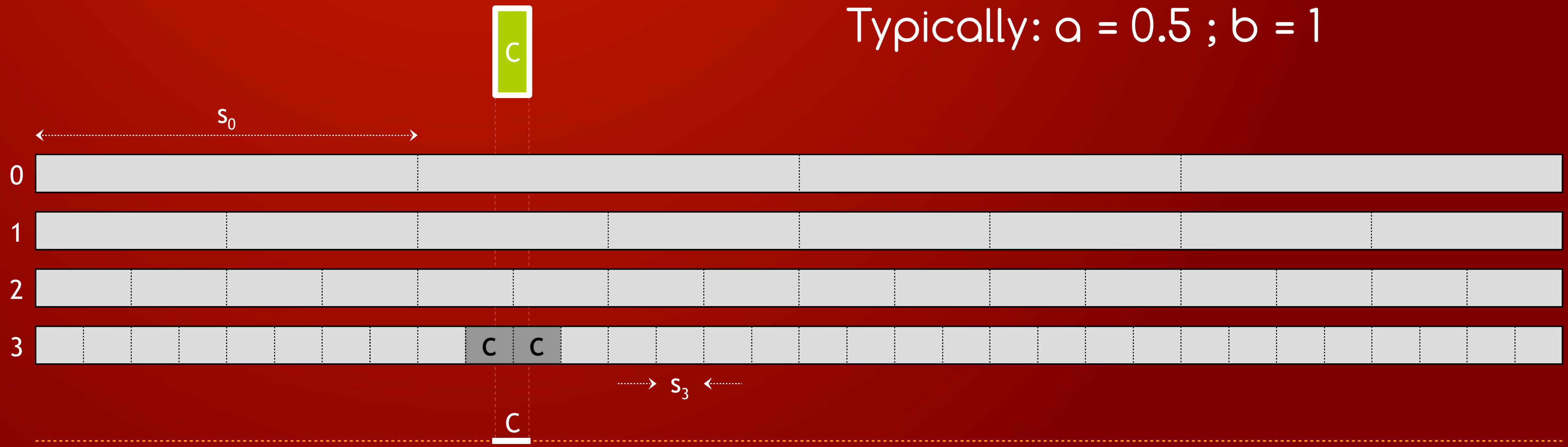
- Suppose 4 uniform grids with 2^k resolutions
  - Grid-0: cell size $s_0 = 1/2^0 = 1.000$
  - Grid-1: cell size $s_1 = 1/2^1 = 0.500$
  - Grid-2: cell size $s_2 = 1/2^2 = 0.250$
  - Grid-3: cell size $s_3 = 1/2^3 = 0.125$



Hierarchical Uniform Grid - Principle

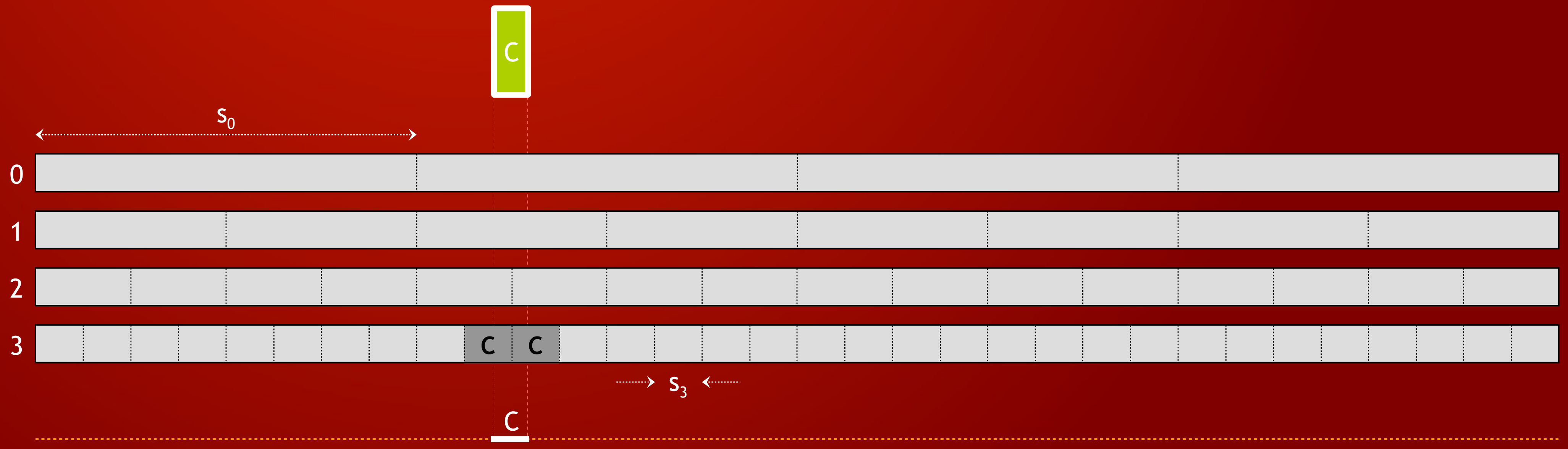
- Find resolution of object "C": Res(C) = 3
  - Cell sizes in grids: $S = (s_0, s_1, \ldots, s_k)$
  - Object box: $AABB(C) = (C_{x-}, C_{y-}, C_{z-}, C_{x+}, C_{y+}, C_{z+})$
  - Object size: $Size(C) = Max(C_{x+} - C_{x+-}, C_{y+} - C_{y+-}, C_{y+} - C_{y+-})$
  - Object resolution: $Res(C) = i \quad \Leftrightarrow \quad a \le (Size(C)/s_i) \le b$

Typically: $a = 0.5$; $b = 1$



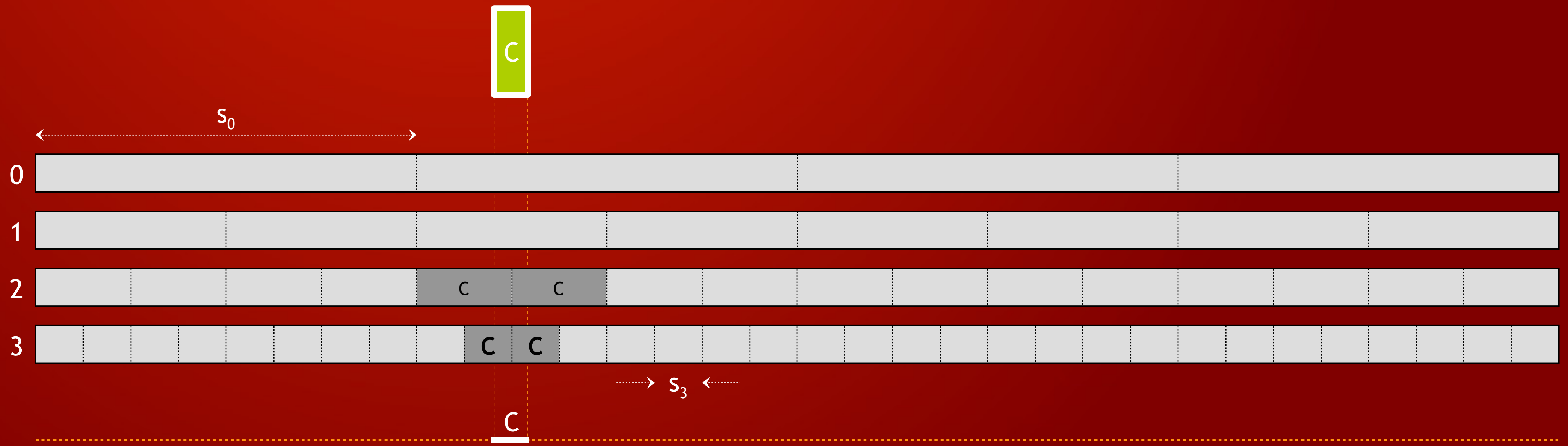
Hierarchical Uniform Grid - Principle

- Insert “C” into grid-3



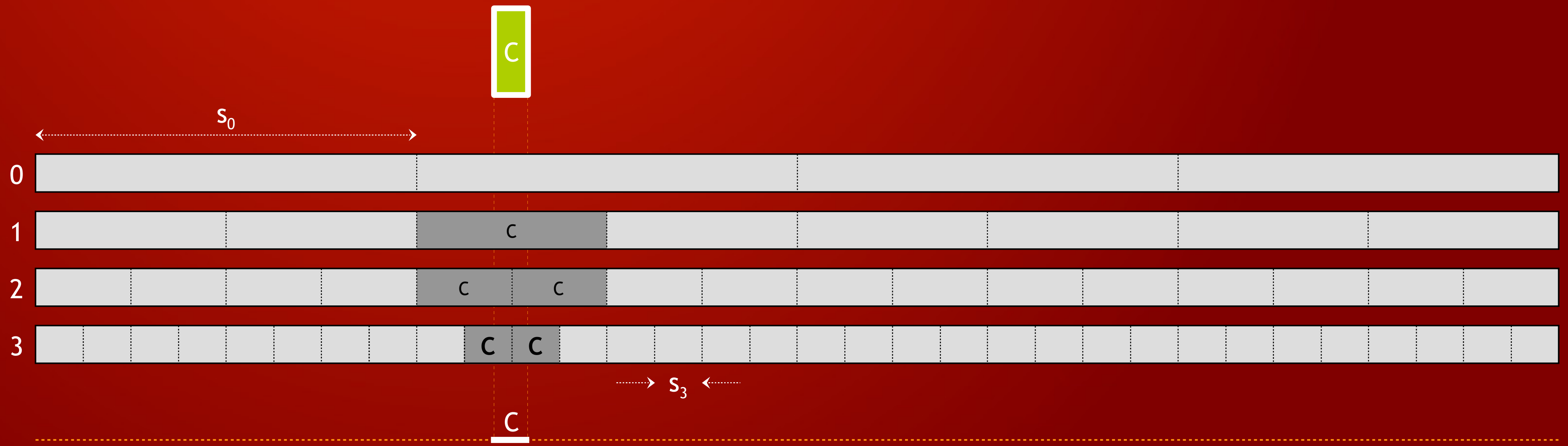
Hierarchical Uniform Grid - Principle

- Insert "C" into grid-3
- Insert "C" into grid-2



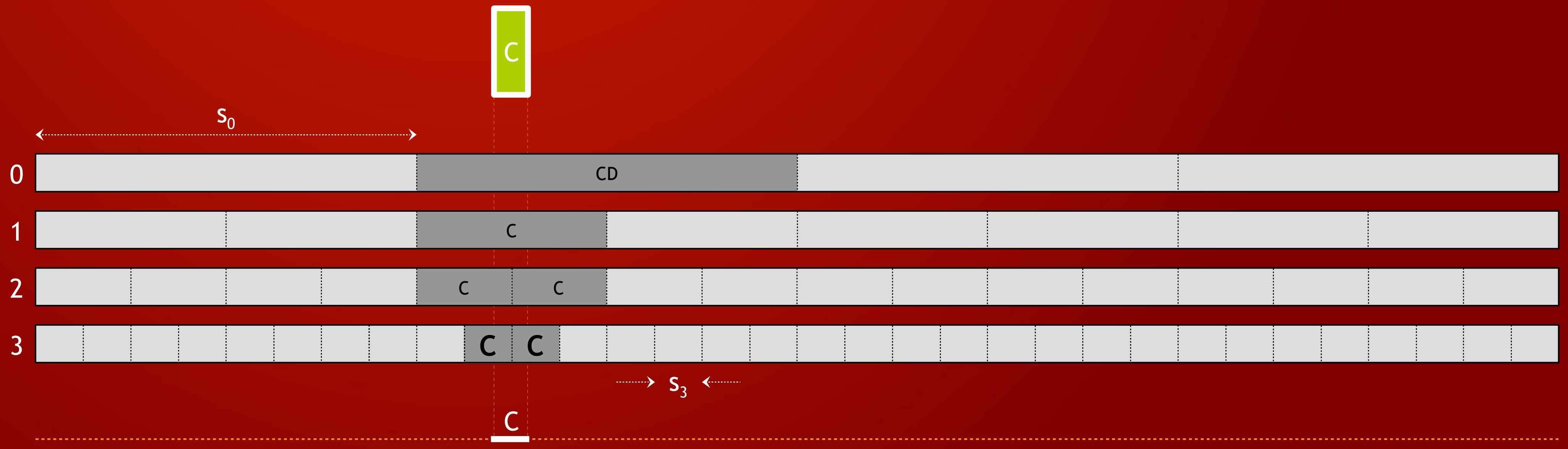
Hierarchical Uniform Grid - Principle

- Insert "C" into grid-3
- Insert "C" into grid-2
- Insert "C" into grid-1



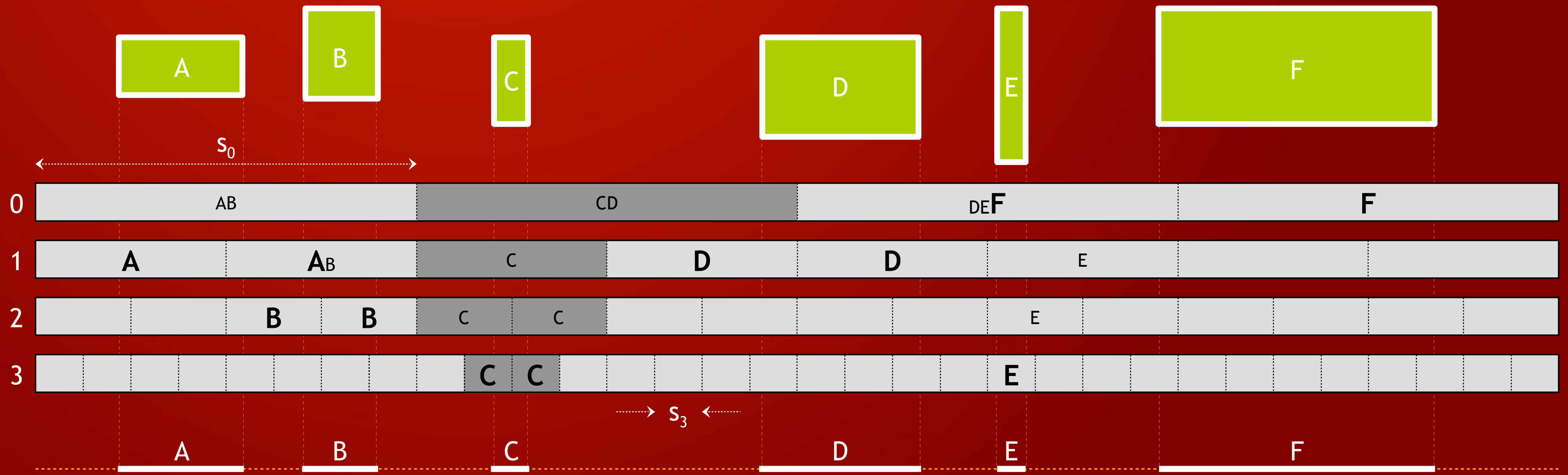
Hierarchical Uniform Grid - Principle

- Insert "C" into grid-3
- Insert "C" into grid-2
- Insert "C" into grid-1
- Insert "C" into grid-0



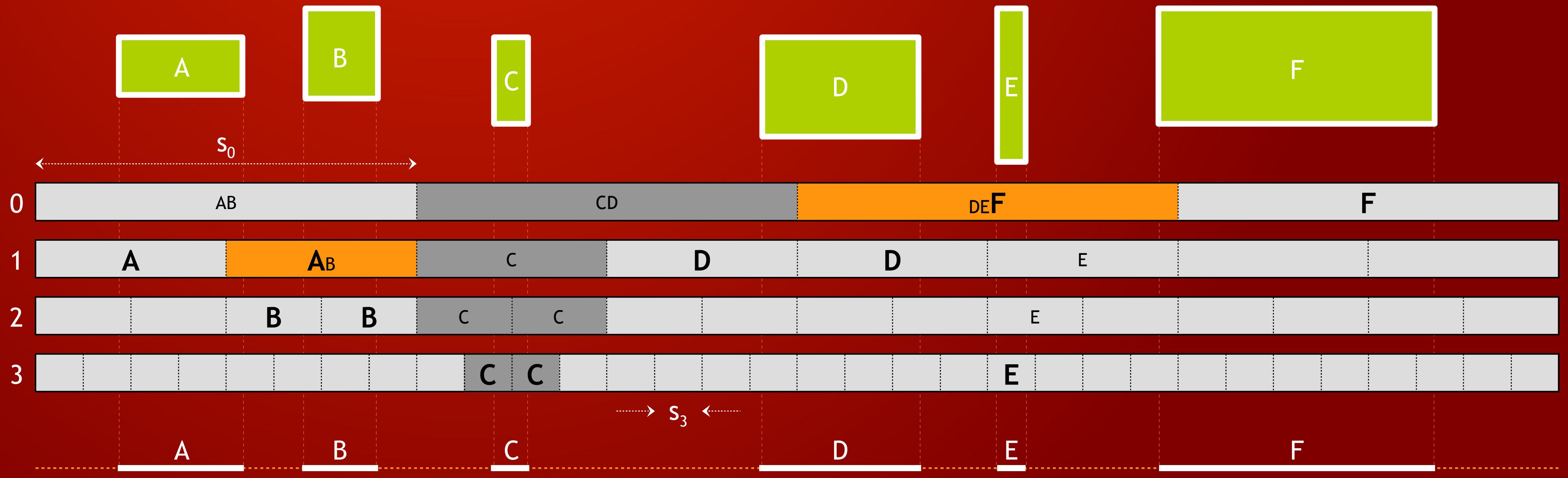
Hierarchical Uniform Grid - Principle

- Insert other objects into grids
- Build ID sets in cells
- Mark IDs “bold” which represent the resolution of object



Hierarchical Uniform Grid - Principle

- During insertion report all ID pairs within each cell which are either "bold" x "regular" or "bold" x "bold" IDs
  - Cell (**A**B) has only one pair: A-B
  - Cell (DE**F**) has pairs: D-F and E-F (D-E is not a pair !)



Hierarchical Uniform Grid - Methods

- AddBox(A)
  - 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))
- RemoveBox
  - Calculate AABB(A), resolution r = Res(A), remove box from all grids (0 to r), remove pair (A-A_k) only if grid resolution is Min(Res(A), Res(A_k))
- UpdateBox
  - Since objects ID are stored only in grids with equal or larger resolutions as Res(A) – no need for optimizing update – simply RemoveBox than AddBox every modified object

Hierarchical Uniform Grid - Summary

- Pros
  - Handle small and large dynamic objects No optimal grid size
  - True linear time broad phase algorithm
- Cons
  - More memory (usually 2 times more)
  - Must update (hash) more grids for each object
  - Accuracy depends on the largest resolution
- Constant Update → Linear time complexity
  - Assuming $R = (s^+ / s^-)$ = largest / smallest AABB size is constant
  - We need $k = \log(R)$ grids – is constant
  - One object marks $O(\log R)$ cells – is constant
  - Add/Remove/Update - are constant → time complexity is $O(n)$

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 spatial hashing function as
- $hash(i,j,k) = (ip_1\ \mathbf{xor}\ jp_2\ \mathbf{xor}\ jp_3)\ \mathbf{mod}\ n$
- Where p_1, p_2, p_3 are large prime numbers and n is the size of hash table
- Hash collision are solved with buckets

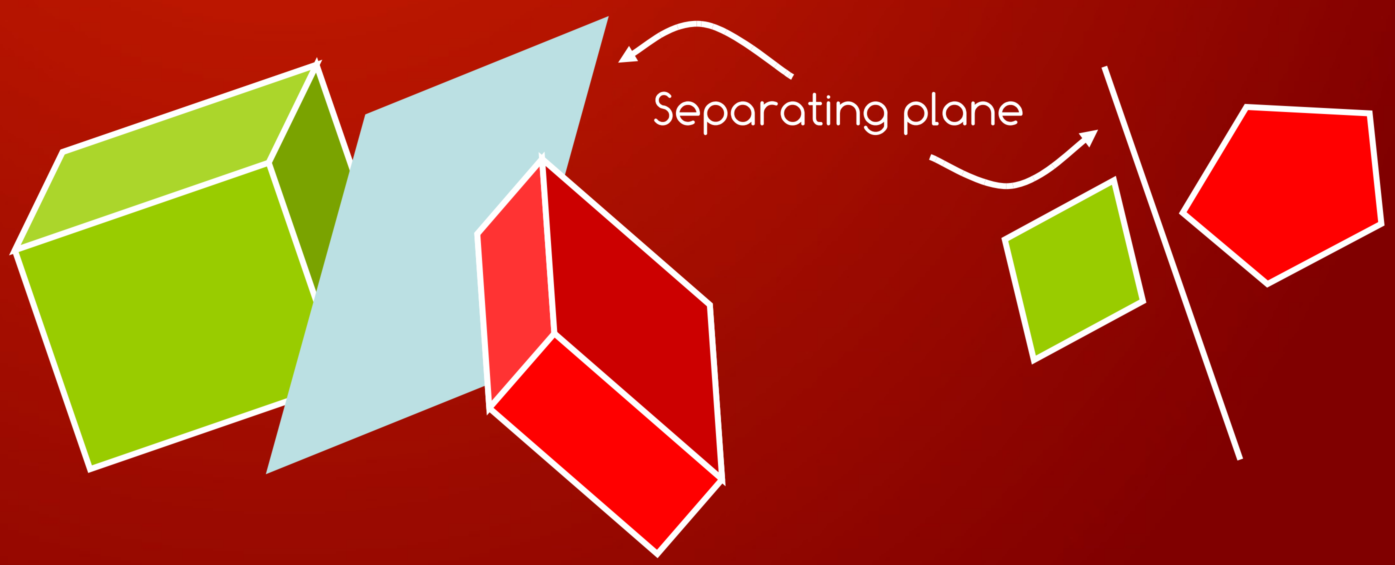

Sweep & Prune

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 prolong objects
  - Too many collisions for high-velocity objects

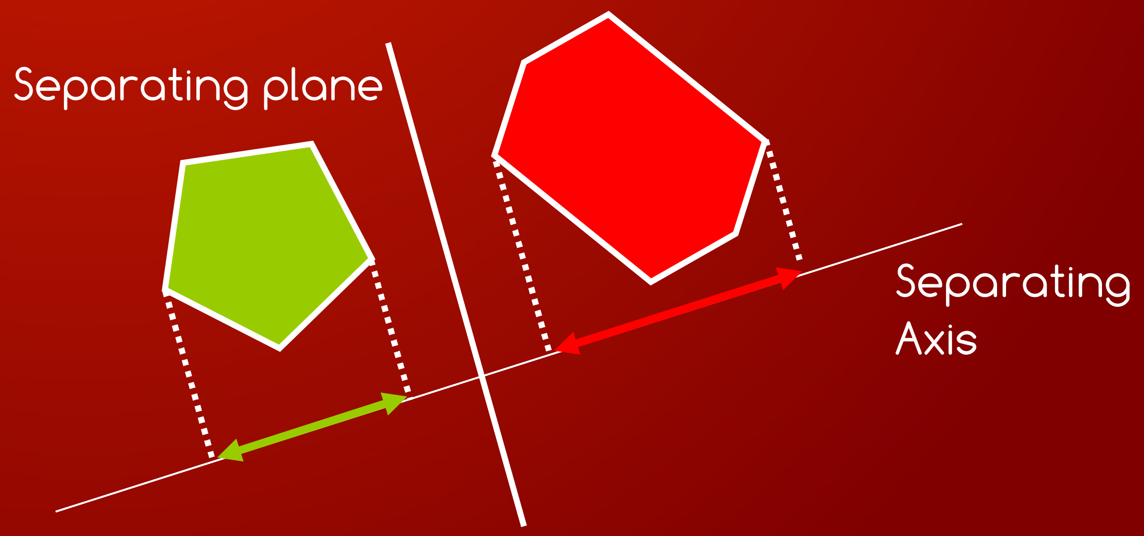
Separating Plane Theorem

- Two convex objects do NOT penetrate (are separated) **if and only if** there exists a (separating) plane which separates them
  - i.e. first (second) object is fully above (below) this plane.



Separating Axis Theorem

- Two convex objects do NOT penetration (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.



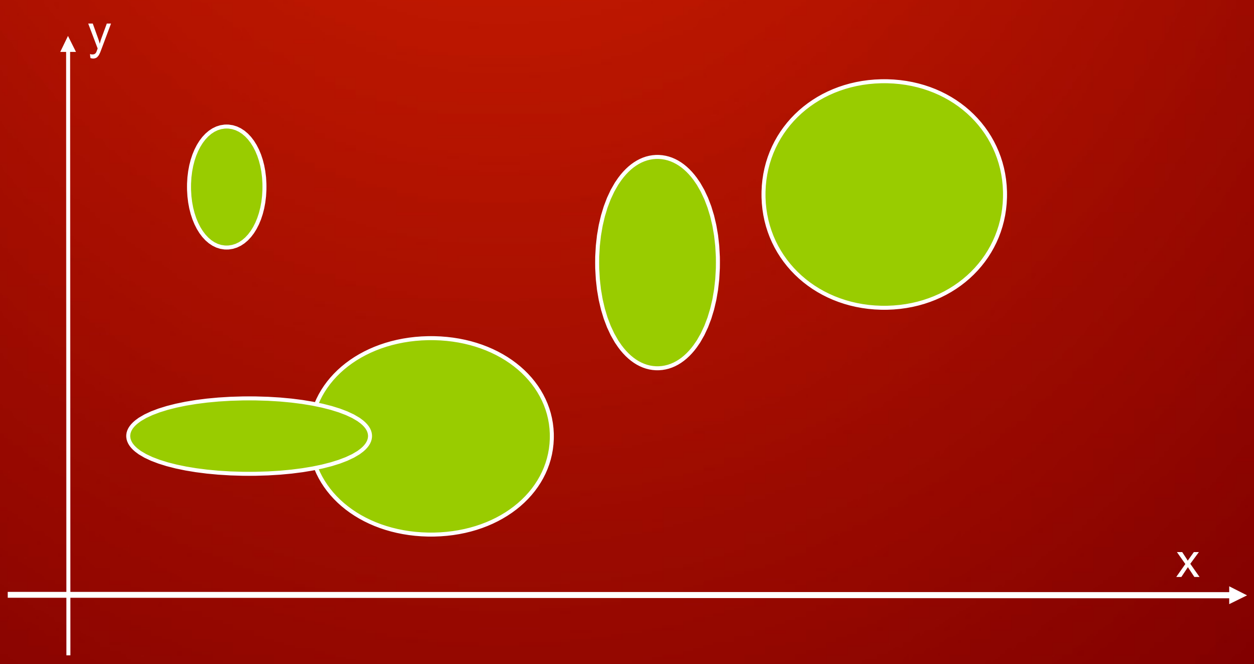
Separating Duality Principle

- For Convex objects
  - Separating Plane Theorem (SPT)
  - Separating Axes Theorem (SAT)
- SAP and STP are equal (dual) !
  - Separating plane and separating axis are perpendicular

SAP ←→ STP

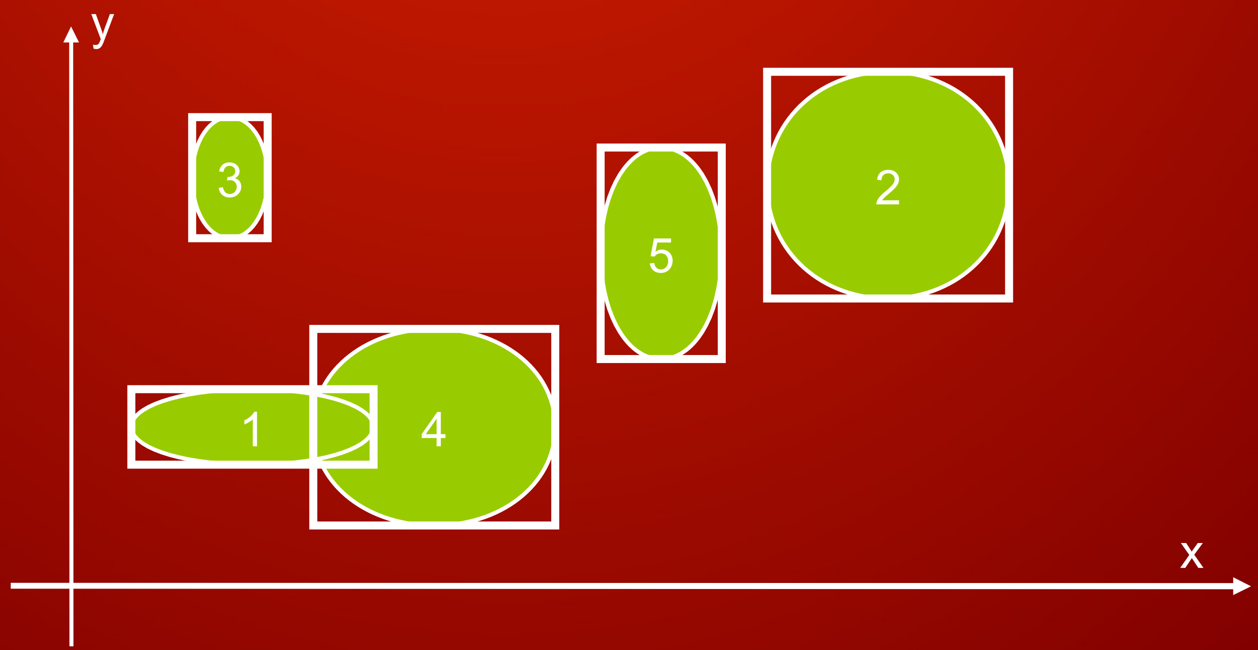
SAP – Algorithm Principle

- Suppose a scene with 5 (not necessarily convex) objects



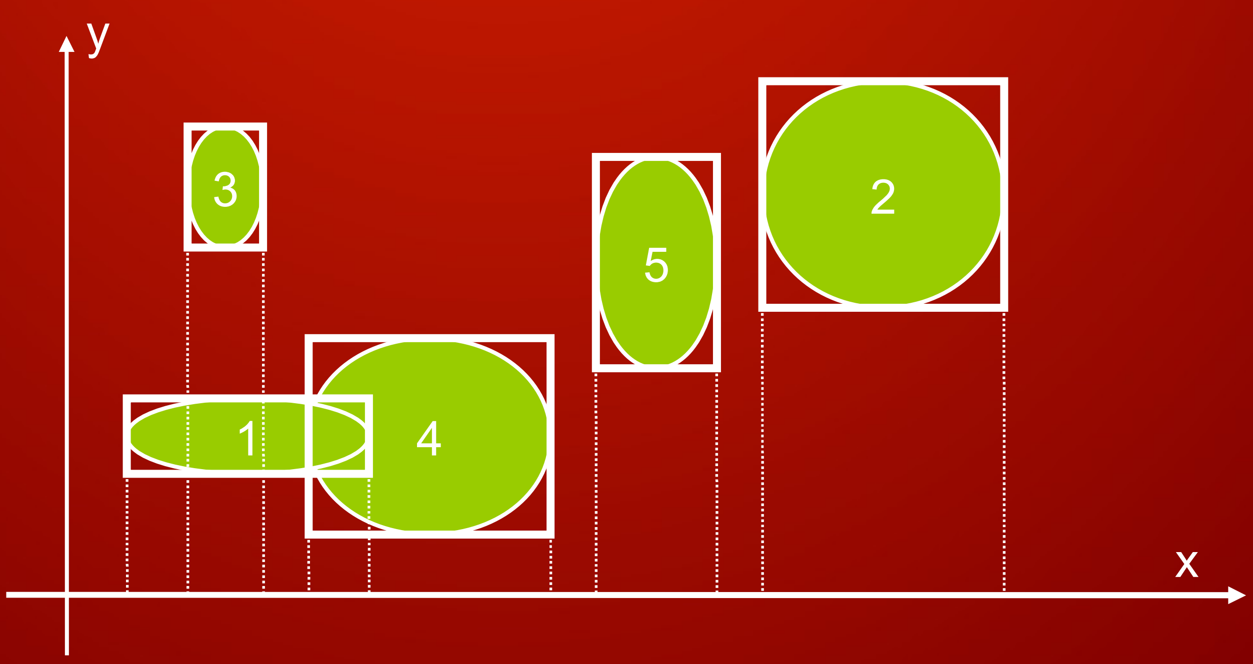
SAP – Algorithm Principle

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



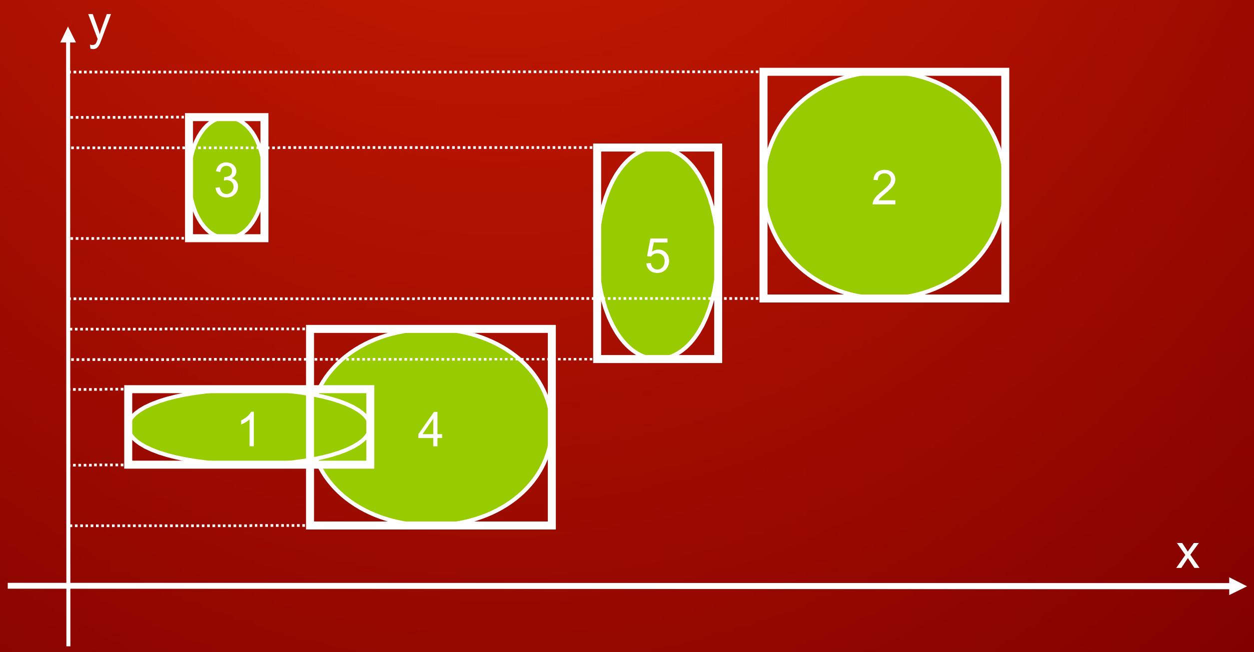
SAP – Algorithm Principle

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



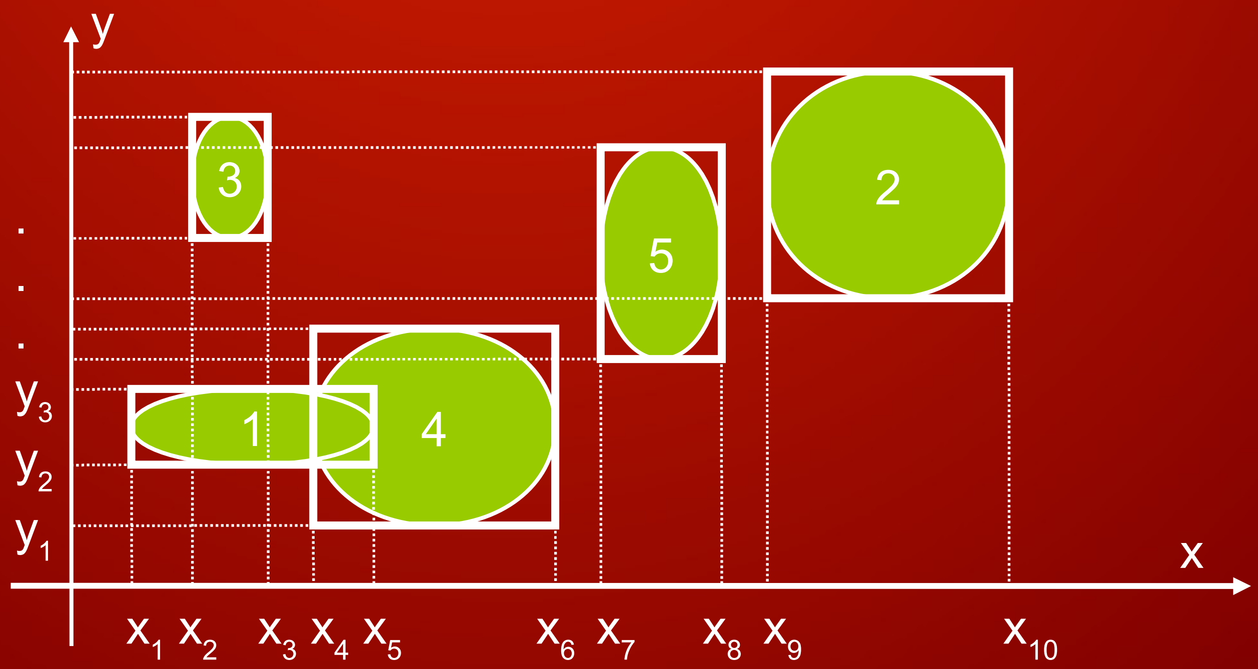
SAP – Algorithm Principle

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



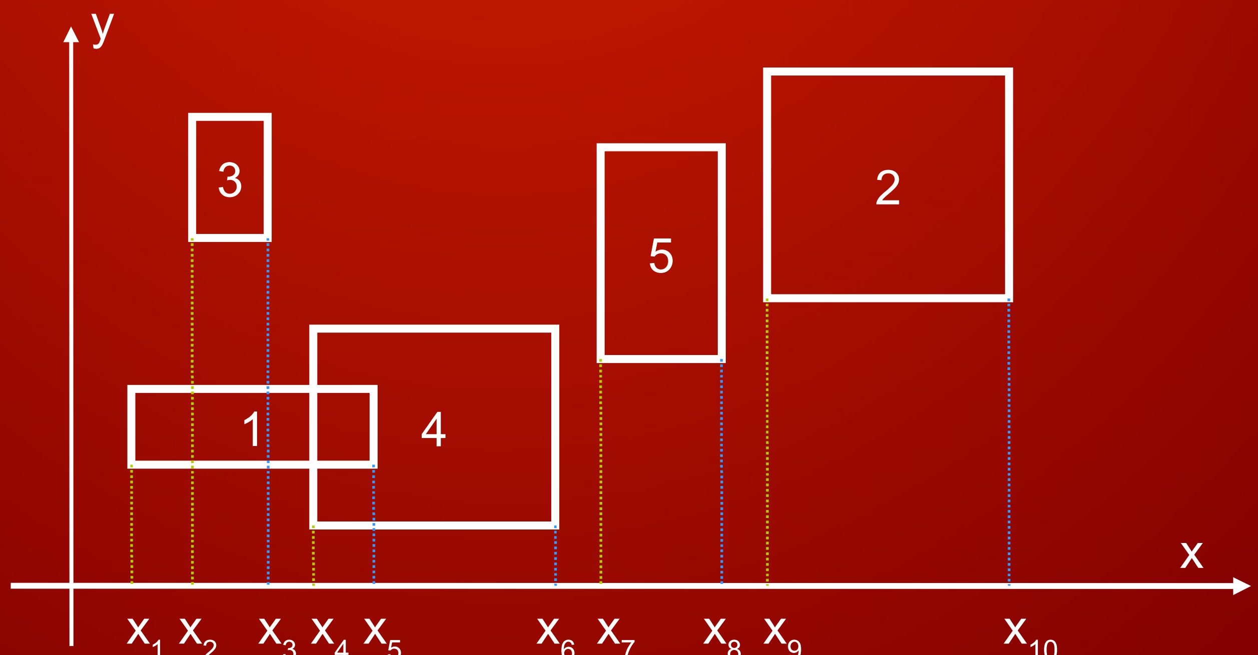
SAP – Algorithm Principle

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



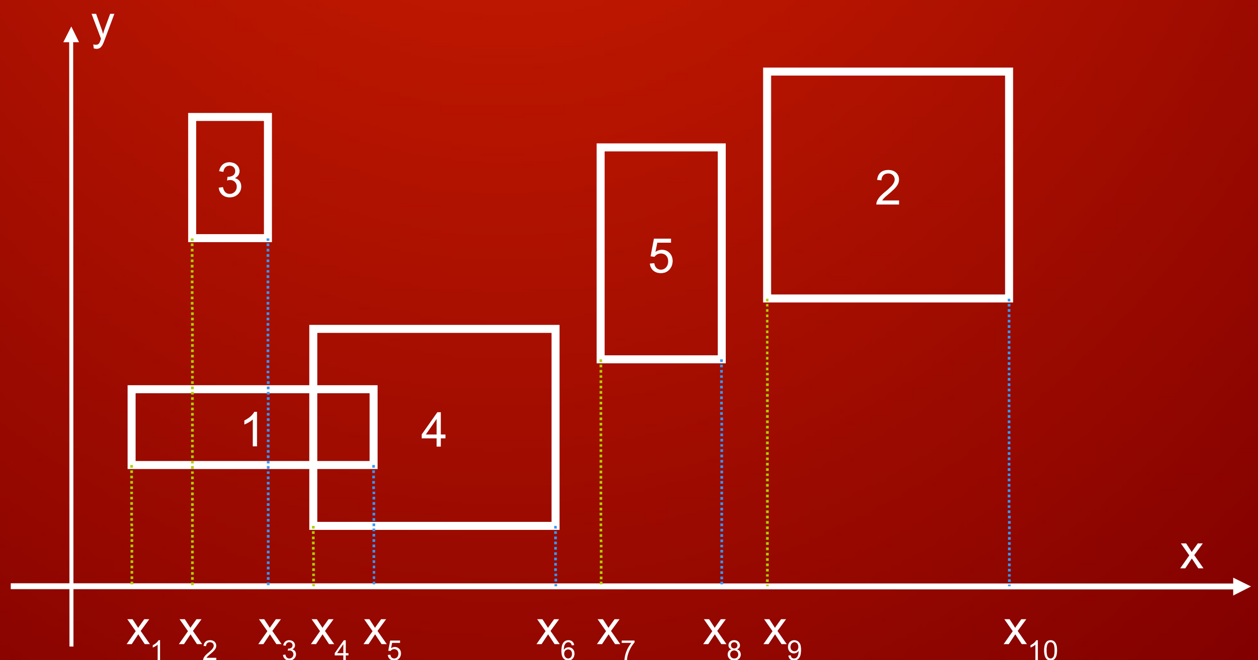
SAP – Algorithm Principle

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



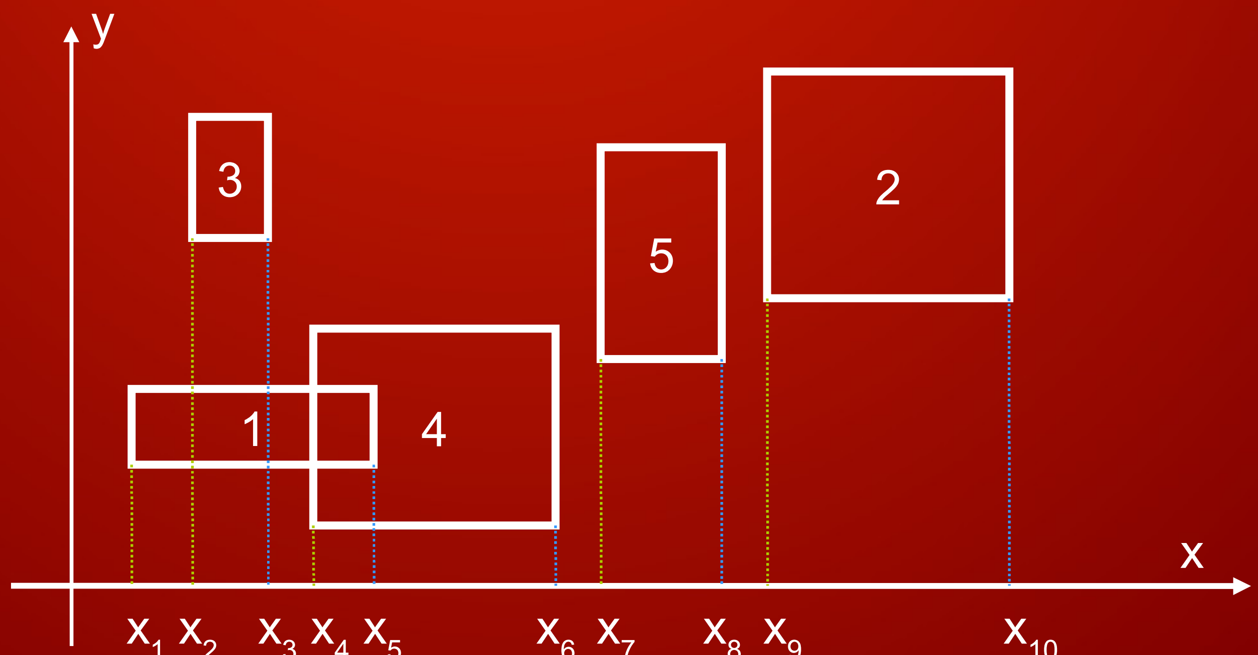
SAP – Algorithm Principle

- 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.



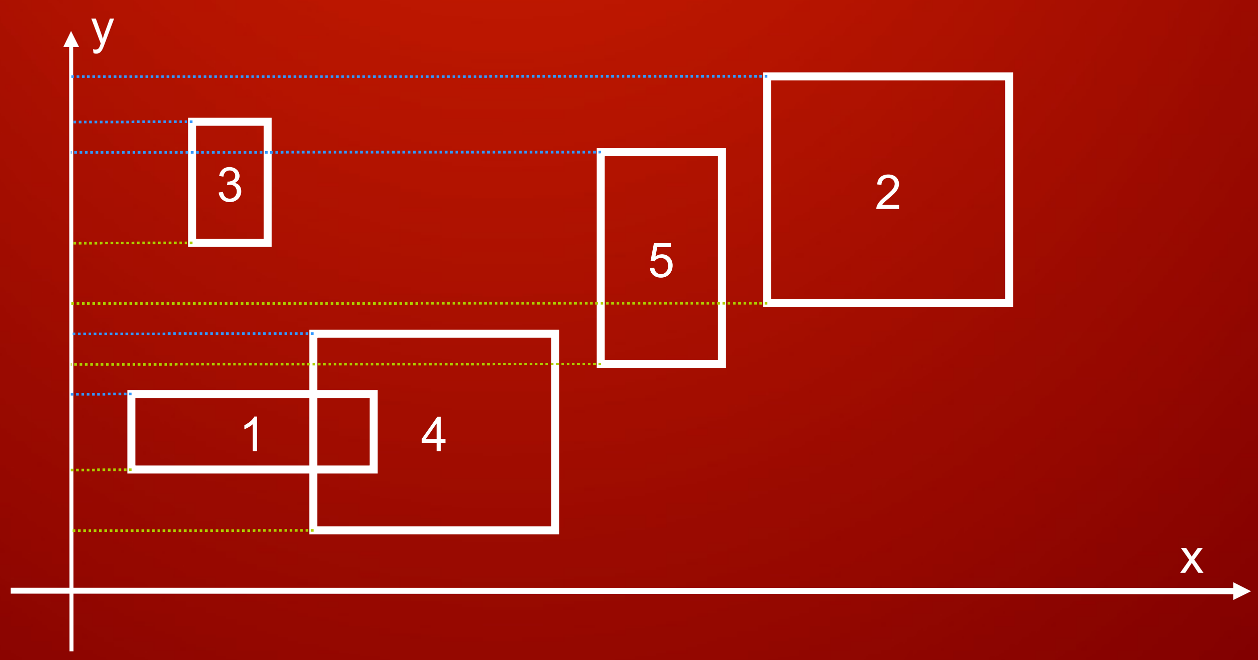
SAP – Algorithm Principle

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



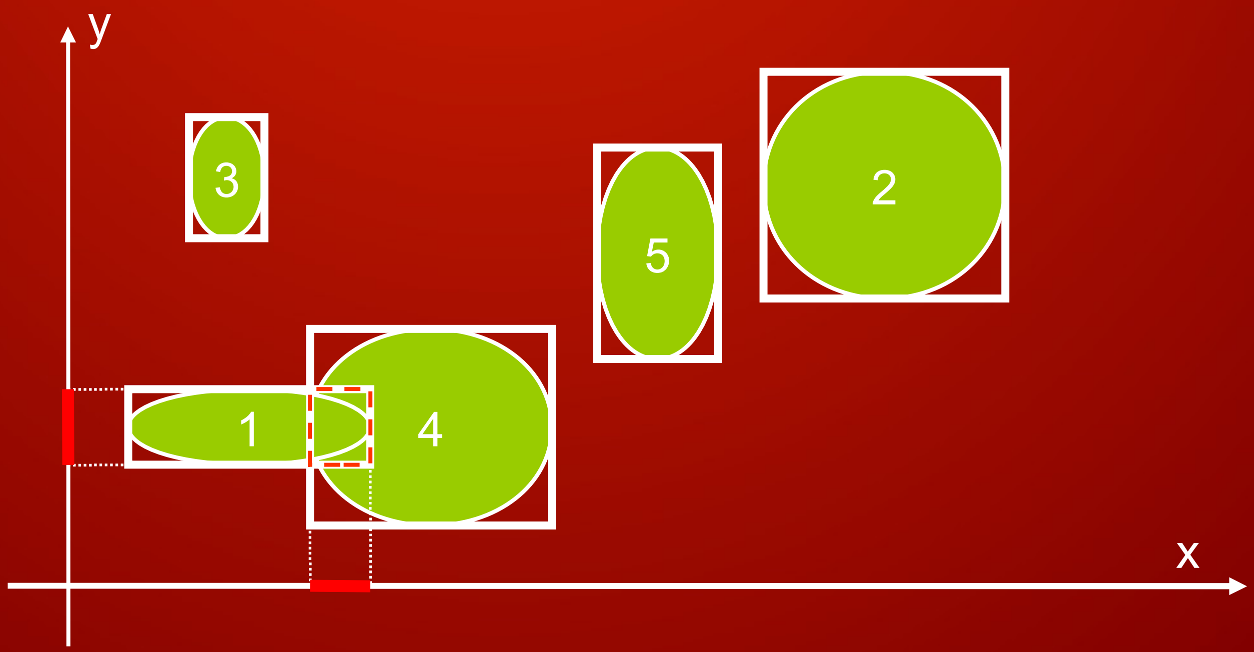
SAP – Algorithm Principle

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



SAP – Algorithm Principle

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

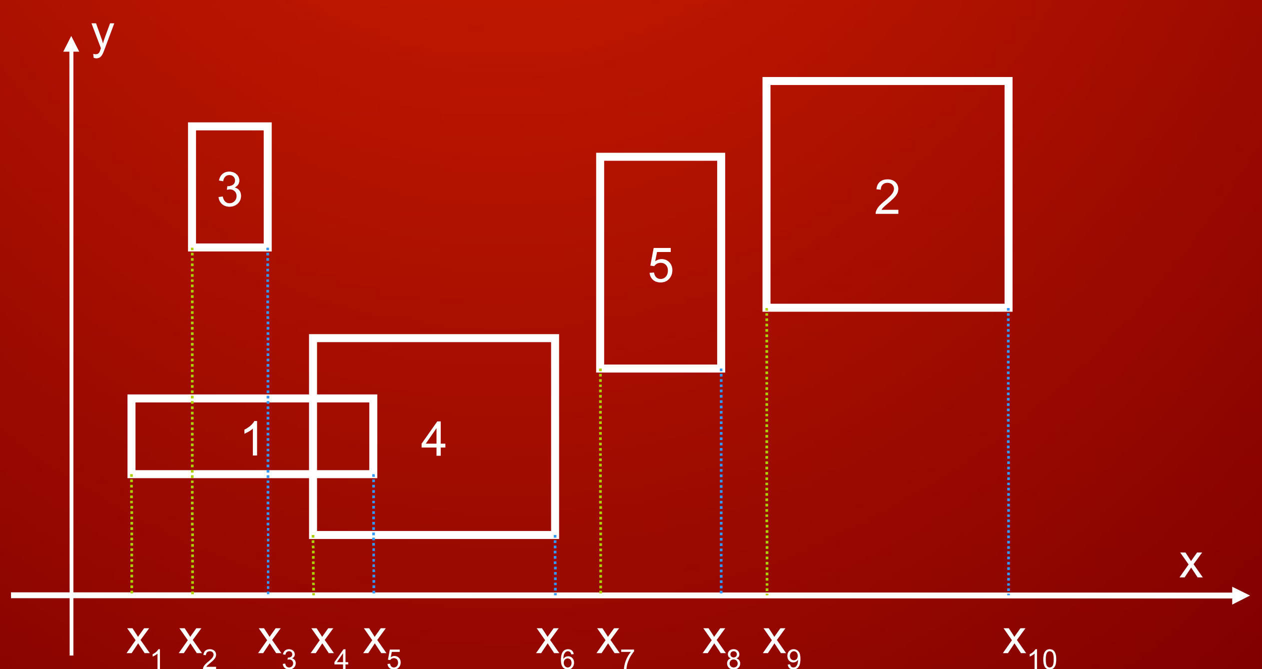


SAP - Summary

- 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.
  - Implement an efficient "pair management" i.e. fast set intersection of axis pair sets (Pairs = SetX ^ SetY ^ SetZ)
- Problems
  - Since objects tend to settle down (usually along Z-axis) during the simulation, large interval clustering can happen

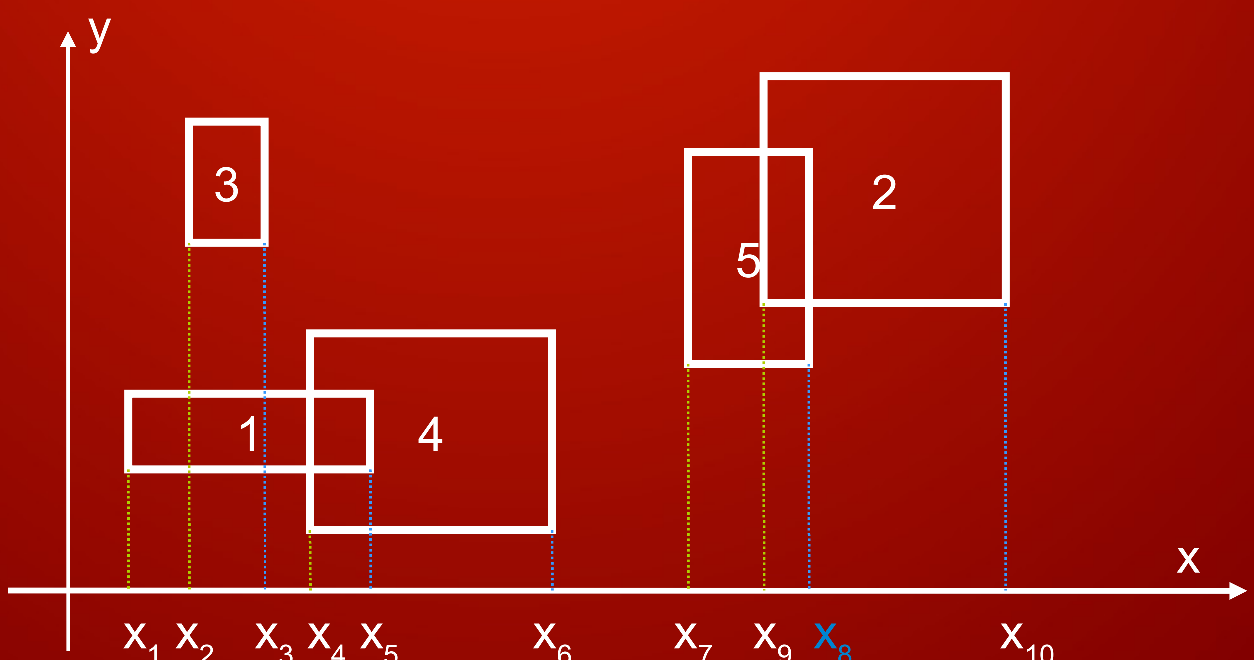
SAP – Incremental Update

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



SAP – Incremental Update

- Reported pairs: (1-3) and (1-4)
- Suppose object 5 moves right
- End limit x_8 pass over x_9 breaking the order
- In this case we report new pair (2-5)

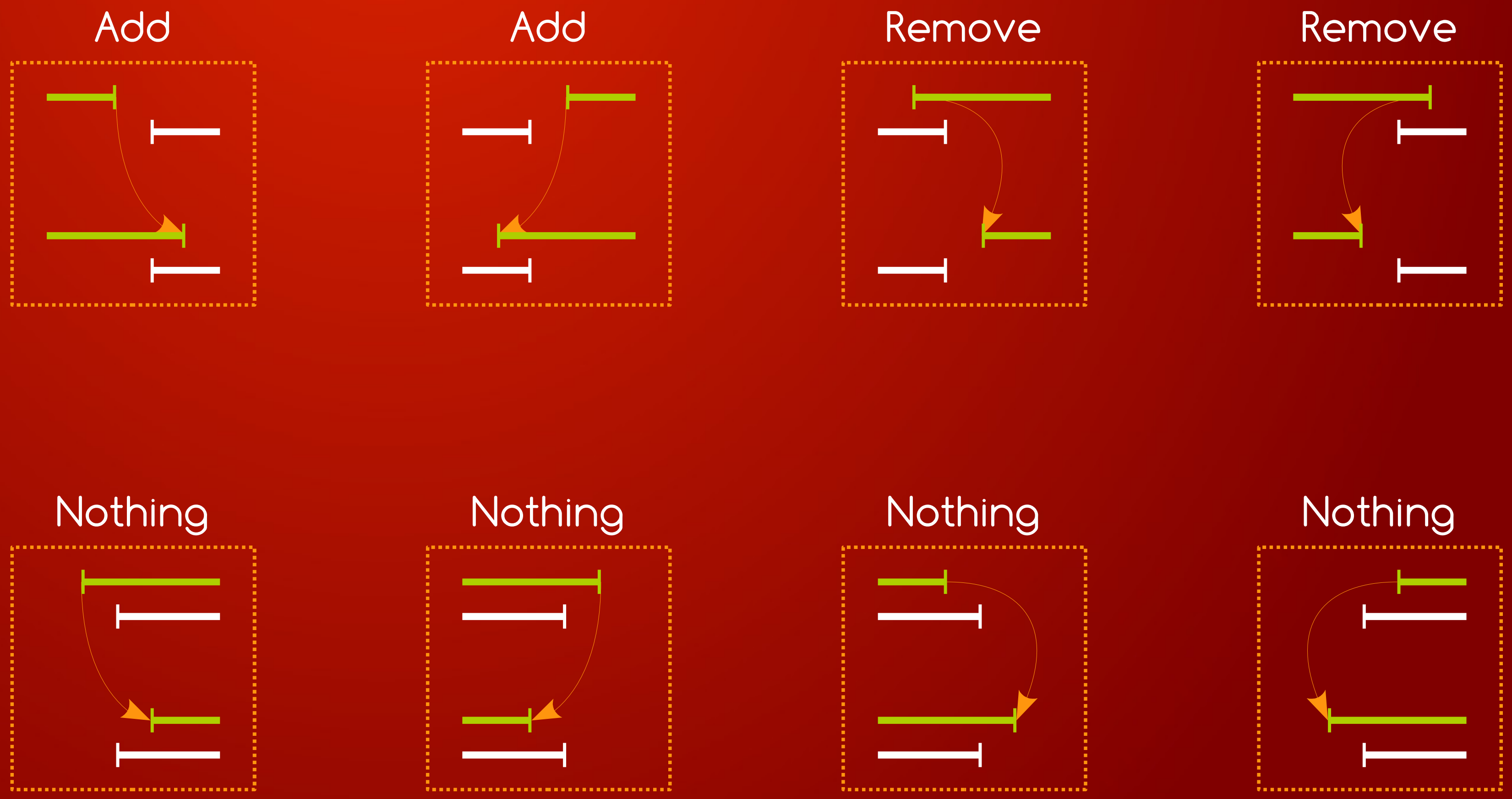


SAP – Incremental Update

- Select moving objects and update theirs 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

SAP – Incremental Update

- Limit swap cases




Pair Management
a practical guide

Pair Management

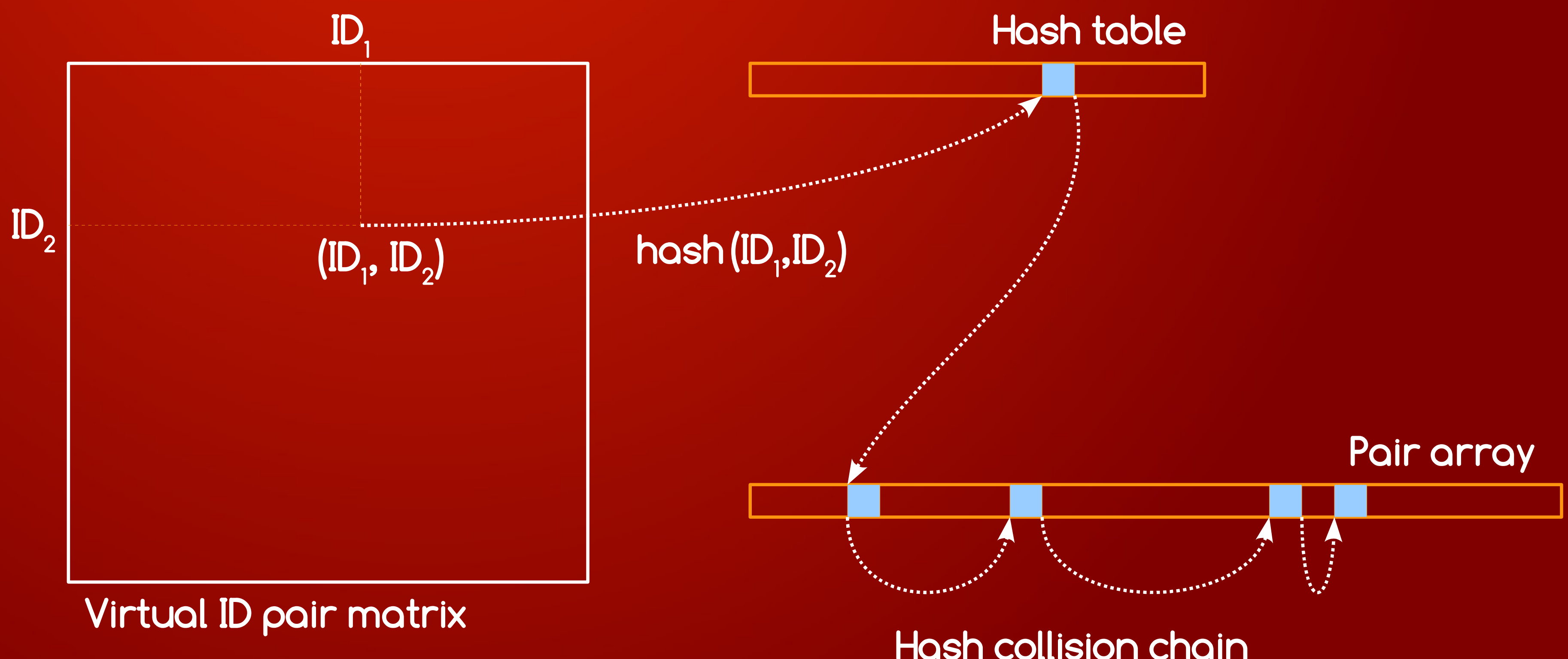
- An ID pair is defined as (ID_1, ID_2)
- Pair Manager is a data structure allowing quickly
  - Adding new pair in $O(1)$: AddPair(ID_1, ID_2)
  - Removing an existing pair in $O(1)$: RemovePair(ID_1, ID_2)
  - Finding an existing pair in $O(1)$: FindPair(ID_1, ID_2)
  - Enumerating all pairs in $O(n)$: GetPairs()
- Trivial approach is to use
  - big matrix to store pair infos - just look at (ID_1, ID_2) item
  - simple list to store set of active pairs.
  - Huge amount of memory, pair list update can be slow
  - Can be efficient for < 1000 objects (matrix size 1000^2 !!!)

Efficient Pair Management

- Use spatial (2d) hashing:
  - $h = hash(ID_1, ID_2) = (ID_1 * p1 + ID_2 * p2) \bmod N$
- Use array bag structure to hold pairs
  - Preallocate "capacity" of data (usually 2 x length)
  - AddPair – stores new pair at the end of array (can resize)
  - RemovePair – move last pair to the removed index – fill the hole
- Point from hash table to pair list
- Chain pairs when hash collision occurs

Efficient Pair Management

- In hash table we store pointer to first pair in the hash collision chain (length k) – should be as small as possible. When k > K (constant) we resize hash table (rehash all pairs). Operations are O(k)=O(1)



Demos / tools / libs

Demos / tools / libs

- Free Open Source Libraries:

- Bullet Physics Library: http://www.bulletphysics.org
  - Bullet collision detection framework
    - http://bulletphysics.org/mediawiki-1.5.8/index.php/CDTestFramework
- Box2D: http://www.box2d.org/
- Chipmunk: http://howlingmoonsoftware.com
- SOFA: http://www.sofa-framework.org/
- Tokamak: http://www.tokamakphysics.com


the
end
that was enough...