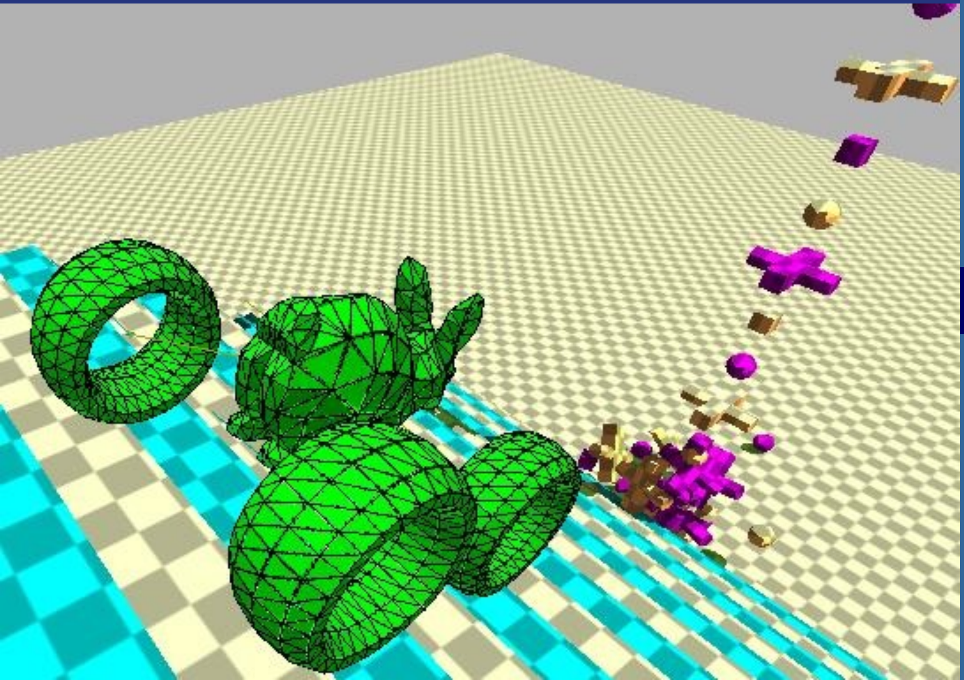
Collision Detection



Originally created by
Tomas Akenine-Möller
Updated by Ulf Assarsson
Department of Computer Engineering
Chalmers University of Technology

What we'll treat today

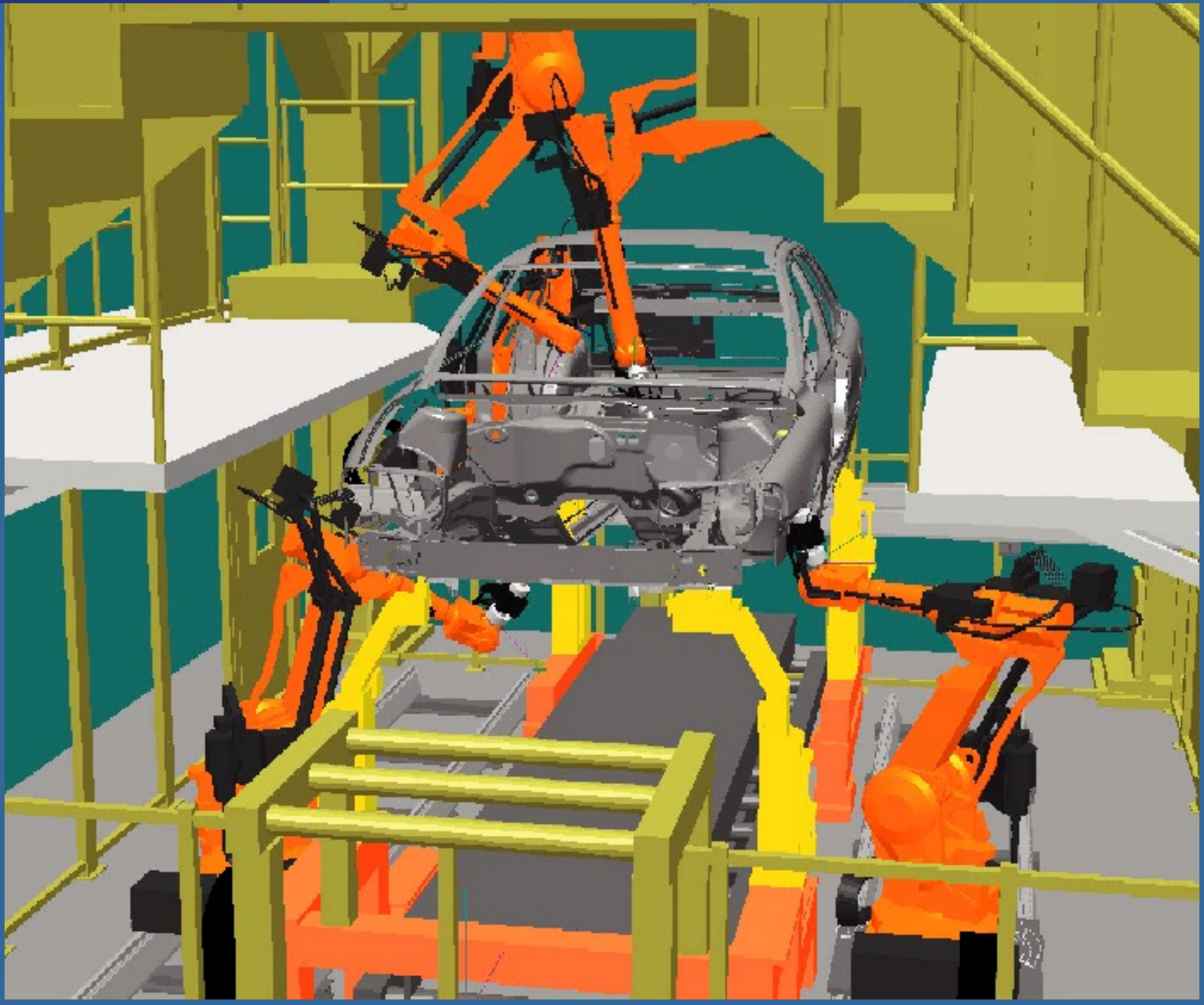
- Three techniques:
- 1) Using ray tracing
  - (Simple if you already have a ray tracer)
  - Not accurate
  - Very fast
  - Sometimes sufficient
- 2) Using bounding volume hierarchies
  - More accurate
  - Slower
  - Can compute exact results
- 3) Efficient CD for several hundreds of objects

Using Ray Tracing

Midtown Madness 3, DICE

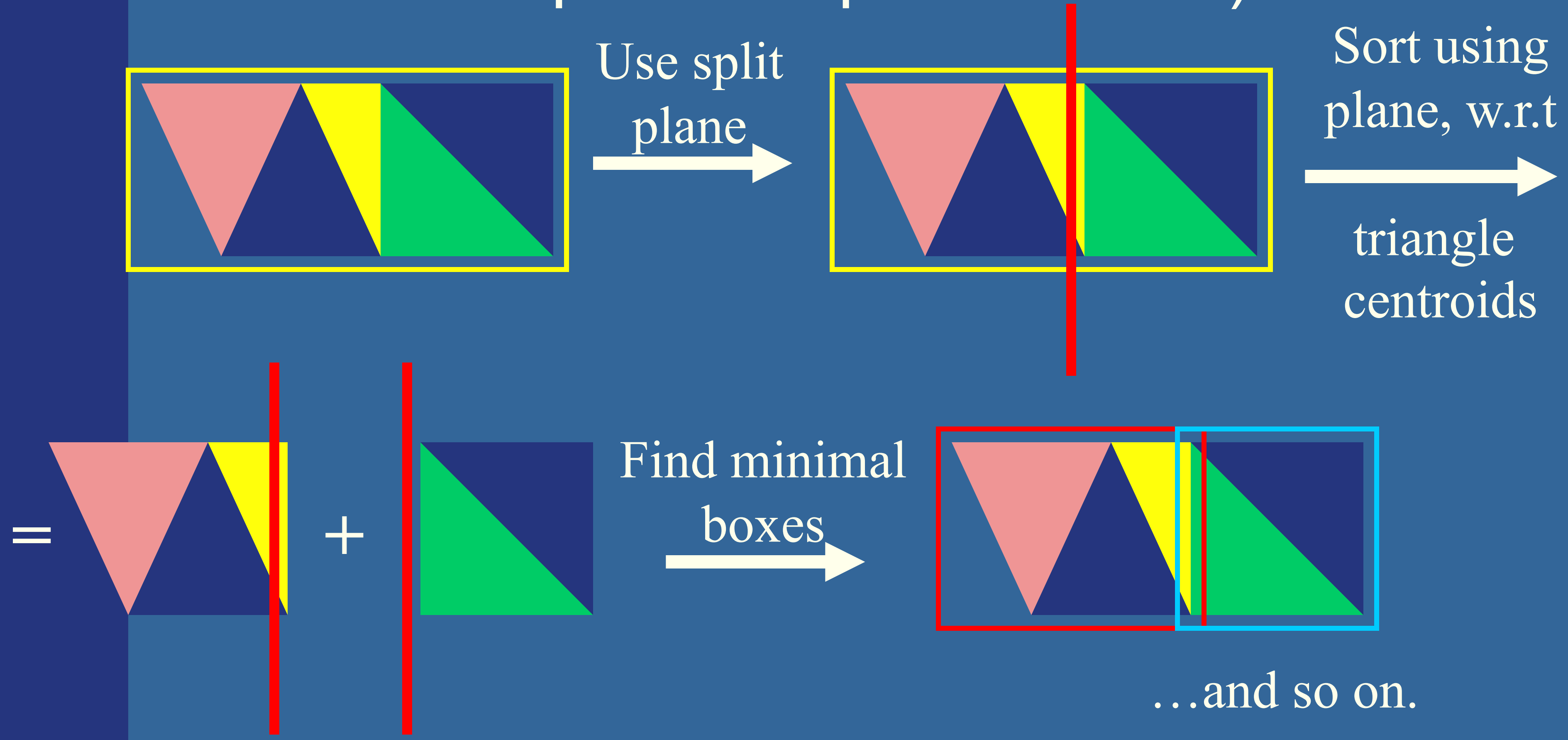
Bounding Volume Hierarchies (BVH)

- If accurate result is needed, turn to BVHs:
  - Use a separate BVH per object
  - Test BVH against other BVH for overlap
    - For all intersecting BV leaves
      - Use triangle-triangle intersection test
- For primitive against primitive CD, see http://www.realtimerendering.com/int/
- But, first, a clarification on BVH building…



BVH building example

- Can split on triangle level as well (not clear from previous presentation)

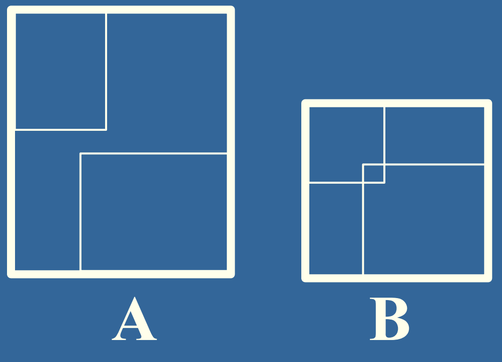


Pseudo code for BVH against BVH

```
FindFirstHitCD(A, B)
if(not overlap(A, B)) return false;
if(isLeaf(A) and isLeaf(B))
   for each triangle pair T_A ∈ A_c and T_B ∈ B_c
      if(overlap(T_A, T_B)) return TRUE;
else if(isNotLeaf(A) and isNotLeaf(B))
   if(Volume(A) > Volume(B))
      for each child C_A ∈ A_c
       if FindFirstHitCD(C_A, B) return true;
   else
      for each child C_B ∈ B_c
       if FindFirstHitCD(A, C_B) return true;
else if(isLeaf(A) and isNotLeaf(B))
   for each child C_B ∈ B_c
    if FindFirstHitCD(C_B, A) return true;
else
   for each child C_A ∈ A_c
    if FindFirstHitCD(C_A, B) return true;
return FALSE;
```

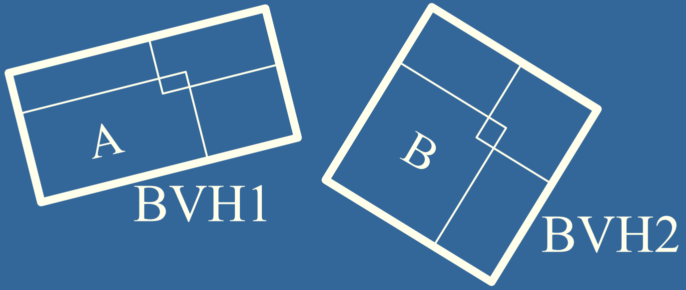
Pseudocode deals with 4 cases:

1) Leaf against leaf node
2) Internal node against internal node
3) Internal against leaf
4) Leaf against internal

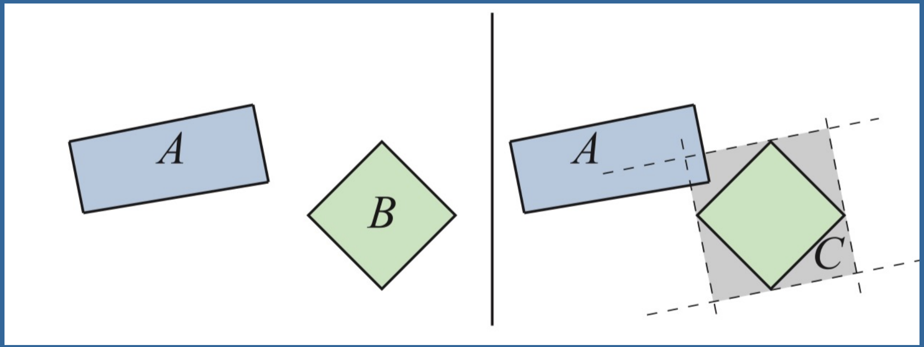


Comments on pseudocode

- The code terminates when it finds the first triangle pair that collides
- Simple to modify code to continue traversal and put each pair in a list, to find all hits.

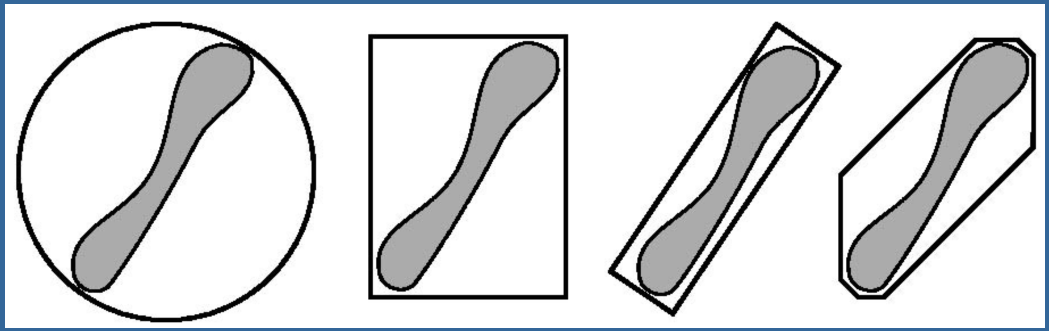


- To handle two AABB hierarchies A, B with different rotations:
  - In overlap(A,B):
    - create an AABB around B in A's coordinate system (below called C). Test A and C against each other
    - And so on, for each node-node test.



Tradeoffs

- The choice of BV
  - AABB, OBB, k-DOP, sphere
- In general, the tighter BV, the slower test



- Less tight BV, gives more triangle-triangle tests in the end

CD for many objects

- Test BV of each object against BV of other object
- Works for small sets, but not very clever
- Reason…
- Assume moving n objects
- Gives: $\frac{n(n-1)}{2}$ tests
- If m static objects, then also $n*m$ tests:
- There are smarter ways…

CD for many objects

- Using Grids:
  - Use a grid with an object list per cell, storing the objects that intersect that cell.
  - For each cell with list length > 1,
    - test the cell's objects against each other using a more exact method (e.g., BVH vs BVH)

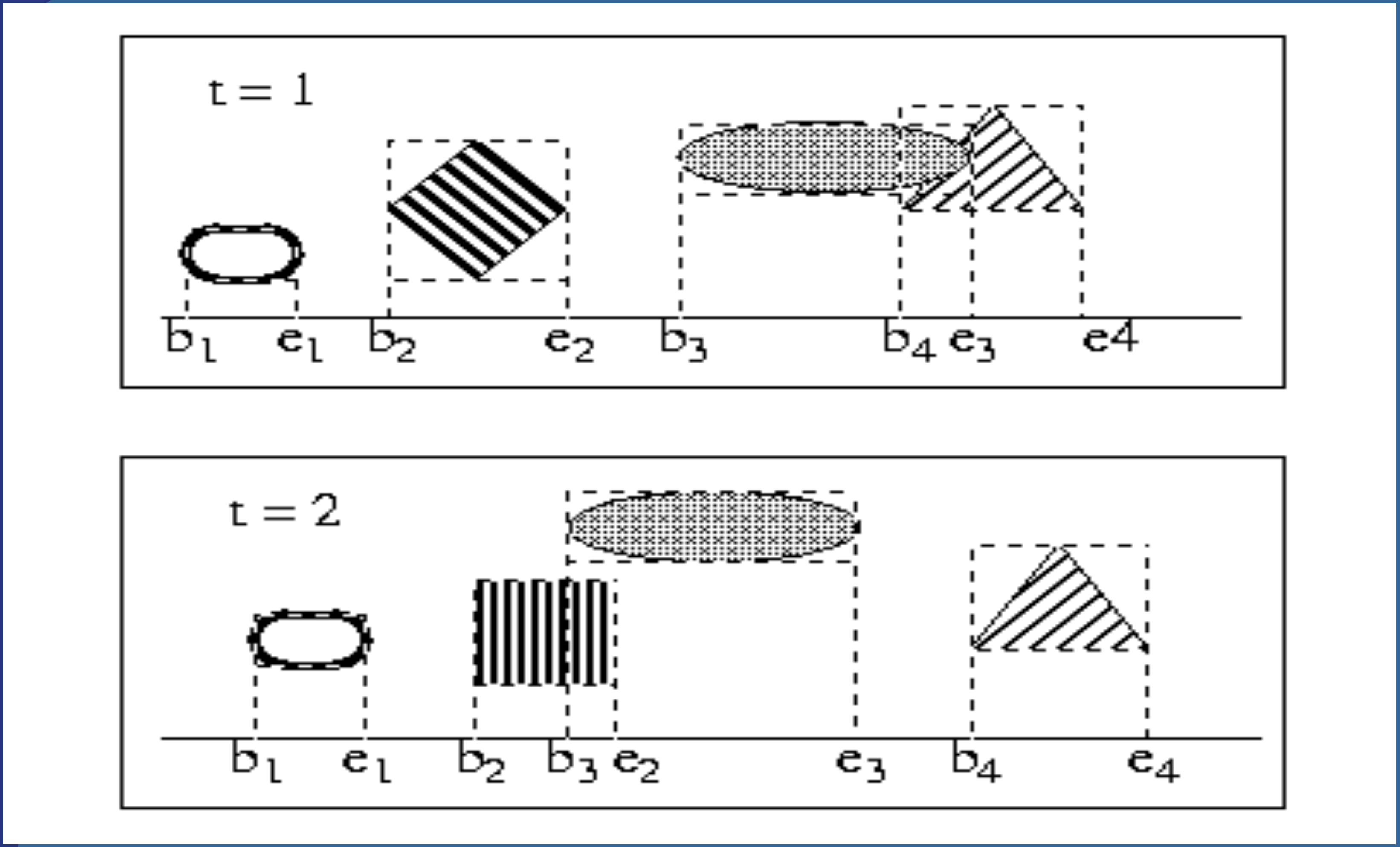
Bonus:

Sweep-and-prune algorithm
[by Ming Lin]

- Assume high frame-to-frame coherency
  - Means that object is close to where it was previous frame
- Do collision overlap three times
  - One for the x,y, and z-axes
- Let's concentrate on one axis at a time
- Each AABB on this axis is an interval, from b_i to e_i, where i is AABB number

Bonus:

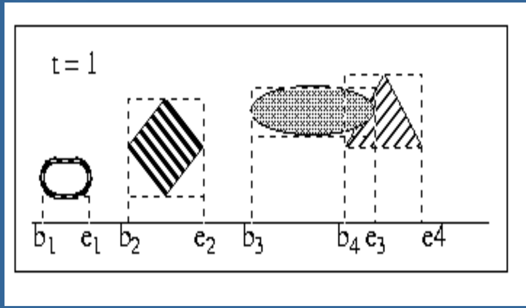
1-D Sweep and Prune



Original by Michael Zyda

Bonus:

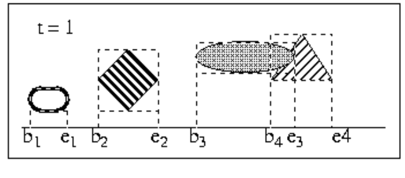
Sweep-and-prune algorithm



- Sort all b_i and e_i into a list
- Traverse list from start to end
- When a b is encounted, mark corresponding object interval as active in an **`active_interval_list`**
- When an e is encountered, delete the interval in **`active_interval_list`**
- All object intervals simultaneously in **`active_interval_ list`** are overlapping on this axis!

Bonus:

Sweep-and-prune algorithm

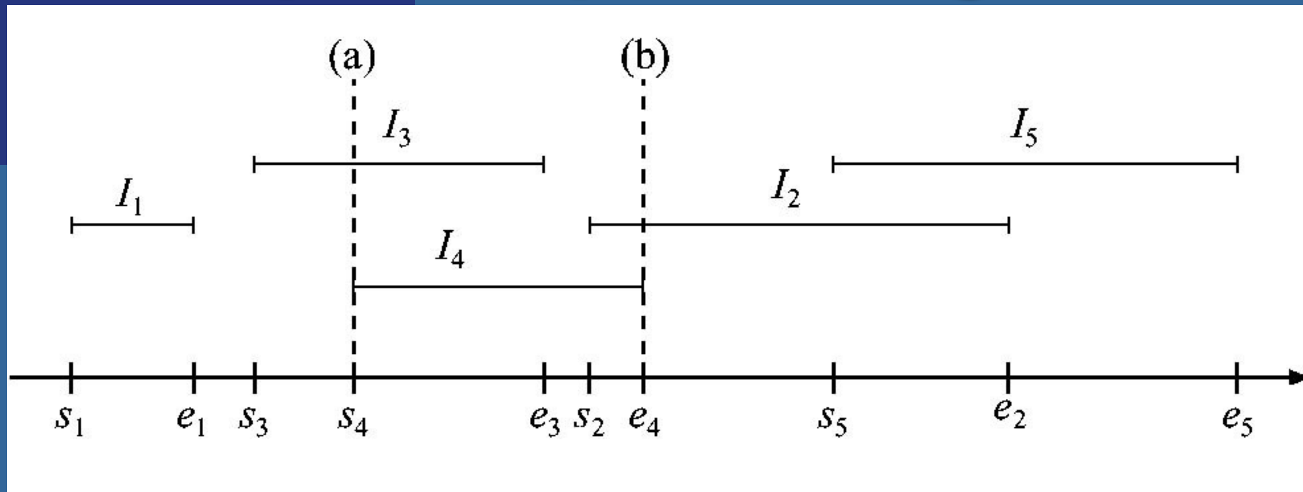


- Now sorting is expensive: O(n*log n)
- But, exploit frame-to-frame coherency!
- The list is not expected to change much
- Therefore, ”resort” with bubble-sort, or insertion-sort
- Expected: O(n)

```
BUBBLE SORT
for (i=0; i<n-1; i++) {
    for (j=0; j<n-1-i; j++)
                //compare the two neighbors
          if (a[j+1] < a[j]) {
               // swap a[j] and a[j+1]
               tmp = a[j];
               a[j] = a[j+1];
               a[j+1] = tmp;
          }
}
```

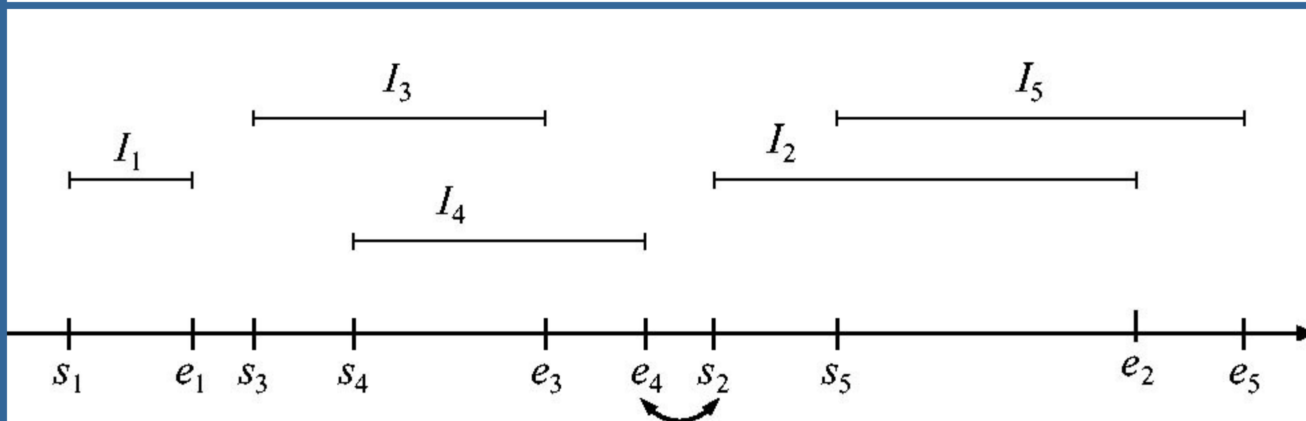
Bonus:

Sweep-and-prune algorithm



	I1	I2	I3	I4
I1		0	0	0
I2			0	1
I3				1
I4				

X axis



	I1	I2	I3	I4
I1		0	0	0
I2			0	0
I3				1
I4				

If (swap(s,e) or swap(e,s)) -> flip bit

- Keep a boolean for each pair of intervals
- Invert boolean when sort order changes
- If all boolean for all three axes are true, → overlap

Bonus:

Efficient updating of the list of colliding pairs (the gritty details)

Only flip flag bit when a start and end point is swapped.

When a flag is toggled, the overlap status indicates one of three situations:

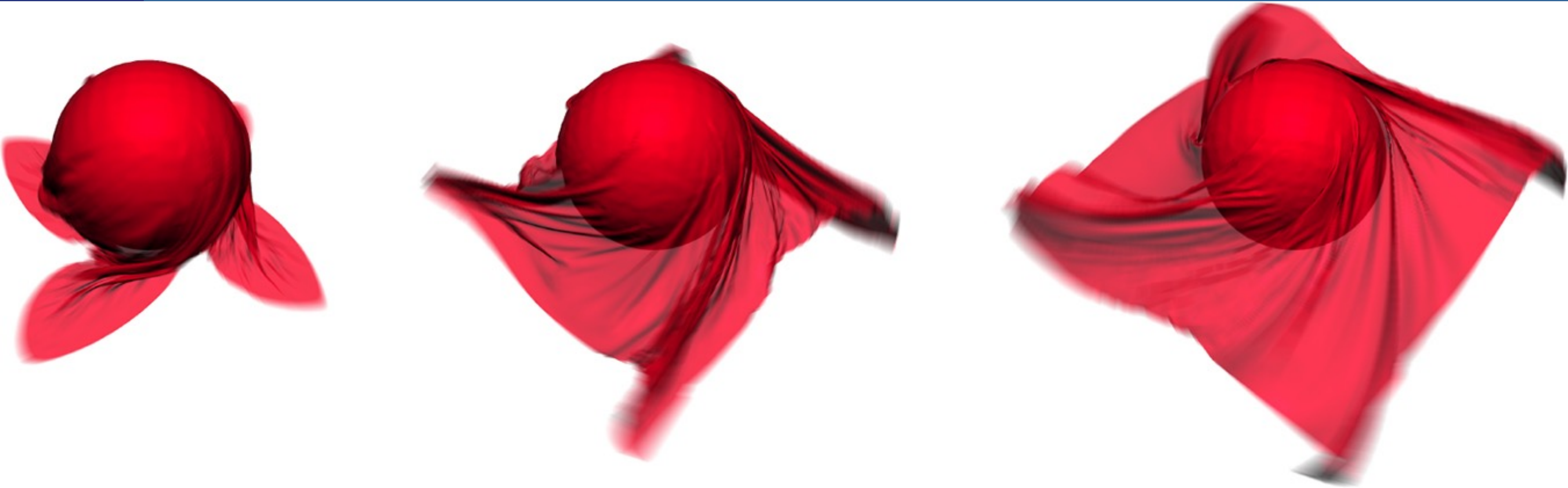
1. All three dimensions of this bounding box pair now overlap. In this case, we add the corresponding pair to a list of colliding pairs.

2. This bounding box pair overlapped at the previous time step. In this case, we remove the corresponding pair from the colliding list.

3. This bounding box pair did not overlap at the previous time step and does not overlap at the current time step. In this case, we do nothing.

Our research

- We use active interval lists per pixel to do correct real-time motion blur with transparency sorting



CD Conclusion

- Very important part of games!
- Many different algorithms to choose from
- Decide what's best for your case,
- and implement…

- **Using Ray tracing vs using BVHs**
- **BVH/BVH-test**
- **Grids**

What you need to know

THE END

- 3 types of algorithms:
  - With rays
    - Fast but not exact (why is it not exact?)
  - With BVH
    - You should be able to write pseudo code for BVH/BVH test for collision detection between two objects.
    - Slower but exact
    - Examples of bounding volumes:
      - Spheres, AABBs, OBBs, k-DOPs
  - For many many objects.
    - pruning of non-colliding objects
    - E.g., Use a grid with an object list per cell, storing the objects that intersect that cell. For each cell with list length > 1, test those objects against each other with a more exact method like BVHs.