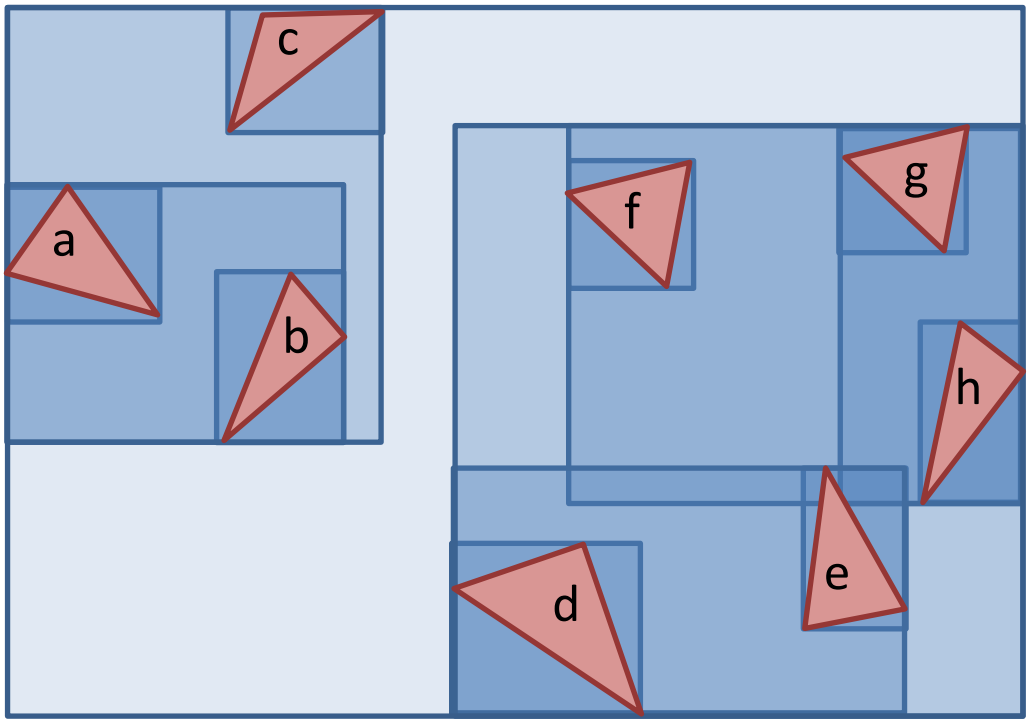
Spatial Data Structures
Part 2

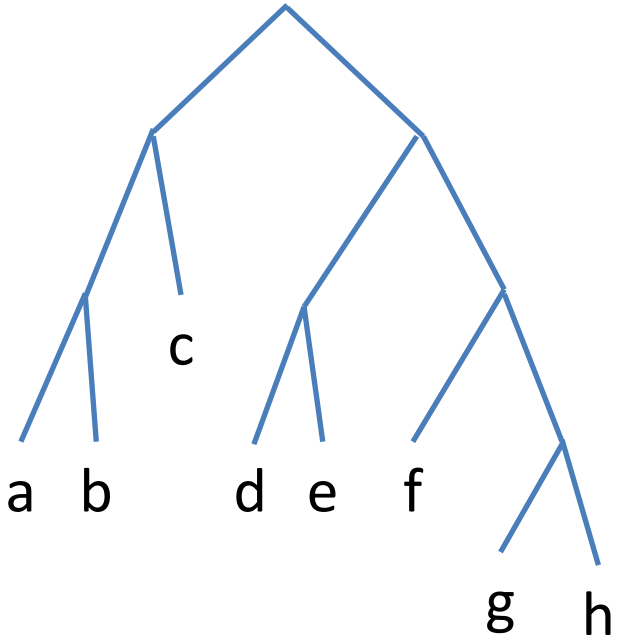
Steve Rotenberg

CSE168: Rendering Algorithms

UCSD, Spring 2014

Axis Aligned Bounding Box Tree





Hierarchical Data Structures

- In general, in order to achieve logarithmic performance, you need some sort of hierarchical (tree) structure
- Searching through N triangles is replaced by searching down log N level of a tree
- In a spatial tree, the levels of the tree represent volumes of space
- The top level represents the volume enclosing all of the triangles
- Each level down the tree splits this volume into smaller sub-volumes that contain a smaller subset of the triangles
- Eventually we get down to *leaf* volumes that contain only one (or a small number usually less than 10) triangles
- The choice of the volumes used is really the main differentiation between the different types of hierarchical data structures we will discuss

Hierarchical Data Structures

- There are two fundamental operations common to all types of hierarchical data structures used in ray tracing:
  - Tree construction
  - Ray traversal

Tree Construction

- The *tree construction* process happens before we begin actual rendering
- We start with a big unorganized array of triangles (sometimes referred to as a *triangle soup*) and construct the tree data structure based on these
- Tree construction algorithms tend to be either top-down or bottom-up
- The standard top-down algorithm starts by fitting a volume around the entire object and then choosing a splitting plane that divides the volume into two groups. Each group then recursively runs the same algorithm, fitting a volume, then splitting into two groups- continuing down to the leaf nodes

Ray Traversal

- The ray traversal process happens for every ray shot into the scene
- We start at the top of the tree and determine if the ray intersects the top volume
- If so, we examine the sub-volumes contained within
- If a sub-volume is intersected, we proceed to test its sub-volumes
- We eventually get down to the leaf nodes and test against the triangles contained
- Because we are interested in the first triangle a ray intersects, we would like to proceed through the volume in a way where we are testing the leaf volumes in the order that the ray intersects them. This way, once we find an intersection in a leaf node, we can be certain that we are done and that there aren't any closer intersections

Spatial Data Structures

- Hierarchical data structures
  - Bounding Volume Hierarchies (BVH)
    - Axis Aligned Box (AABB Tree, Box Tree)
    - Spheres
  - Spatial Partitions
    - K-D Tree (volumes split by x, y, or z plane)
    - BSP Tree (volumes split by arbitrary plane)
    - Nested Grid
    - Octree (nested grid with N=2)
- Non-hierarchical data structures
  - Grid

AABB-Trees

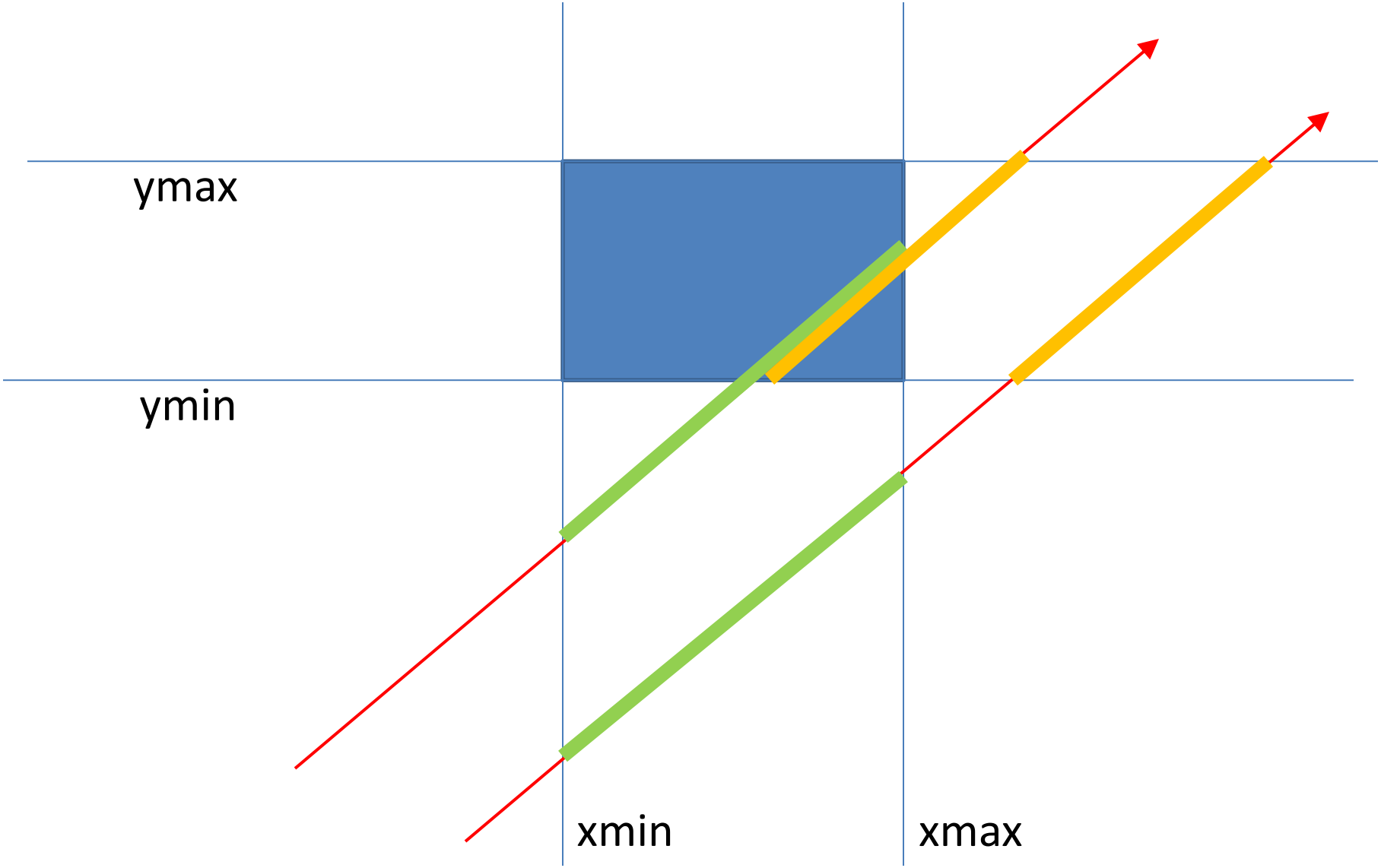
- *Axis Aligned Bounding Box Trees* are also called *AABB trees* or just *box trees*.
- At each node in the tree, we have an axis aligned box represented by 6 numbers (the x, y, z coordinates of the min and max corners)
- Typically, AABB trees are *binary trees* in that each node has zero or two children (however this isn't strictly required for an AABB tree)
- At the leaf nodes, the box contains a small number of triangles- possibly limited to only one, or possibly limited to a small number typically less than 10

Ray Traversal

Ray-AABB Intersection

- To test a Ray against an Axis Aligned Box, we test the ray against the 3 sets of parallel planes, and determine the *t* interval (along the ray) where the planes are intersected
- If the intervals from x, y, and z overlap, then the ray intersects the box

Ray-AABB



Ray-AABB Test

$$t_{x1} = \frac{a_x - p_x}{d_x}, t_{x2} = \frac{b_x - p_x}{d_x}$$
$$t_{y1} = \frac{a_y - p_y}{d_y}, t_{y2} = \frac{b_y - p_y}{d_y}$$
$$t_{z1} = \frac{a_z - p_z}{d_z}, t_{z2} = \frac{b_z - p_z}{d_z}$$
$$t_{min} = \max\{\min(t_{x1}, t_{x2}), \min(t_{y1}, t_{y2}), \min(t_{z1}, t_{z2})\}$$
$$t_{max} = \min\{\max(t_{x1}, t_{x2}), \max(t_{y1}, t_{y2}), \max(t_{z1}, t_{z2})\}$$

- If $t_{min} \leq t_{max}$ then the ray intersects the box at $t = t_{min}$ (or at $t = t_{max}$ if $t_{min} < 0$). If $t_{max} < 0$, then the box is behind the ray and it's a miss.
- Note: **a** and **b** are the min and max corners of the box. **p** and **d** are the ray origin and direction

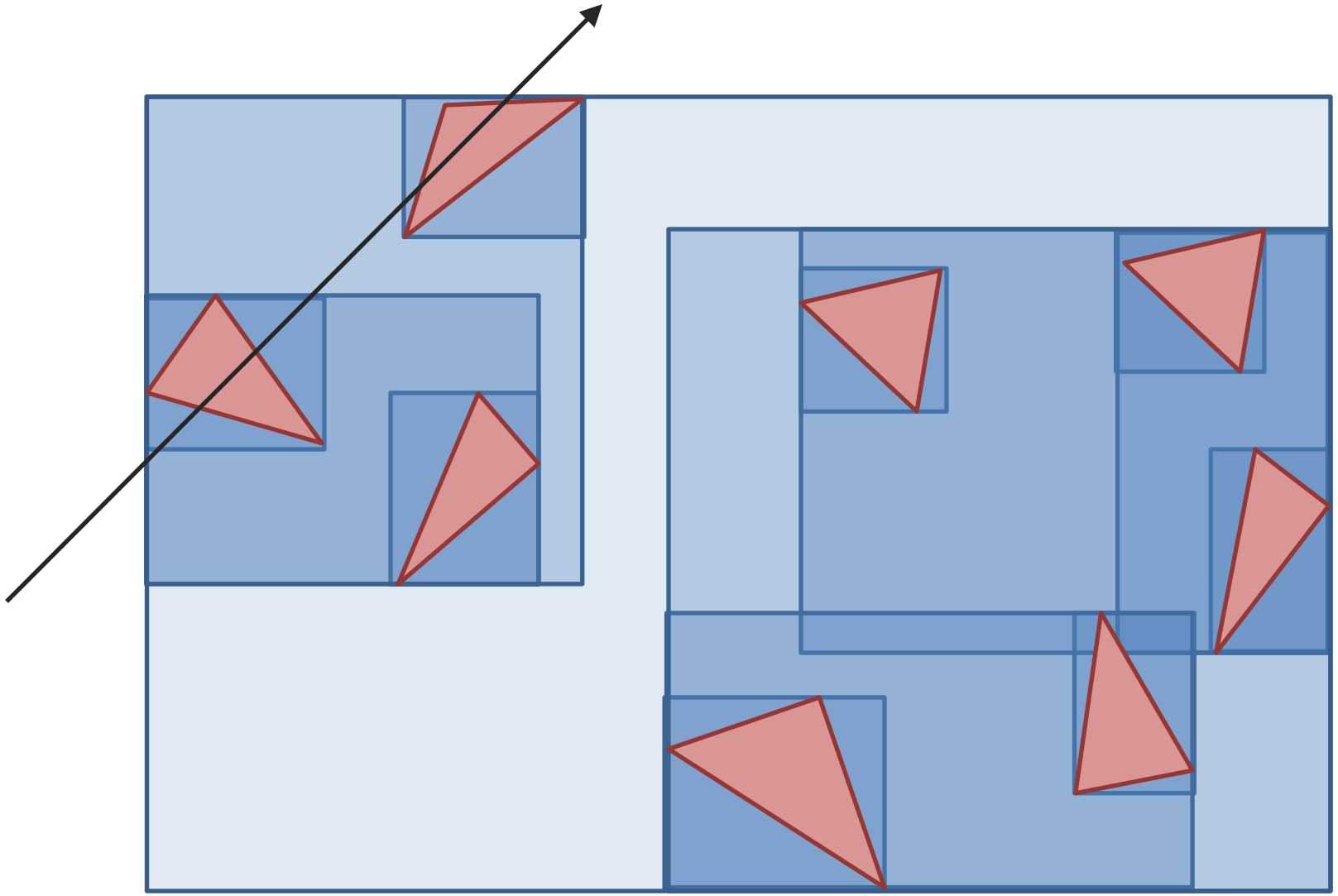
Ray Traversal

- When we ‘traverse’ the data structure, we recursively search down to the leaf volumes along the ray’s direction of travel and examine the triangles contained within
- This mainly involves intersection tests with the boxes themselves, as well as some recursion and bookkeeping
- For a spatial partition, if we test volumes strictly in the order of the ray direction, then once we find a volume with a hit, then we are done
- For a BVH, we have the additional potential of an overlapping volume, so we have a little bit of extra testing to be sure we have the first triangle the ray hits
- In either case, when testing a leaf node, we have to test the ray against all of the triangles contained within, and go with the hit closest to the ray origin if we have multiple hits

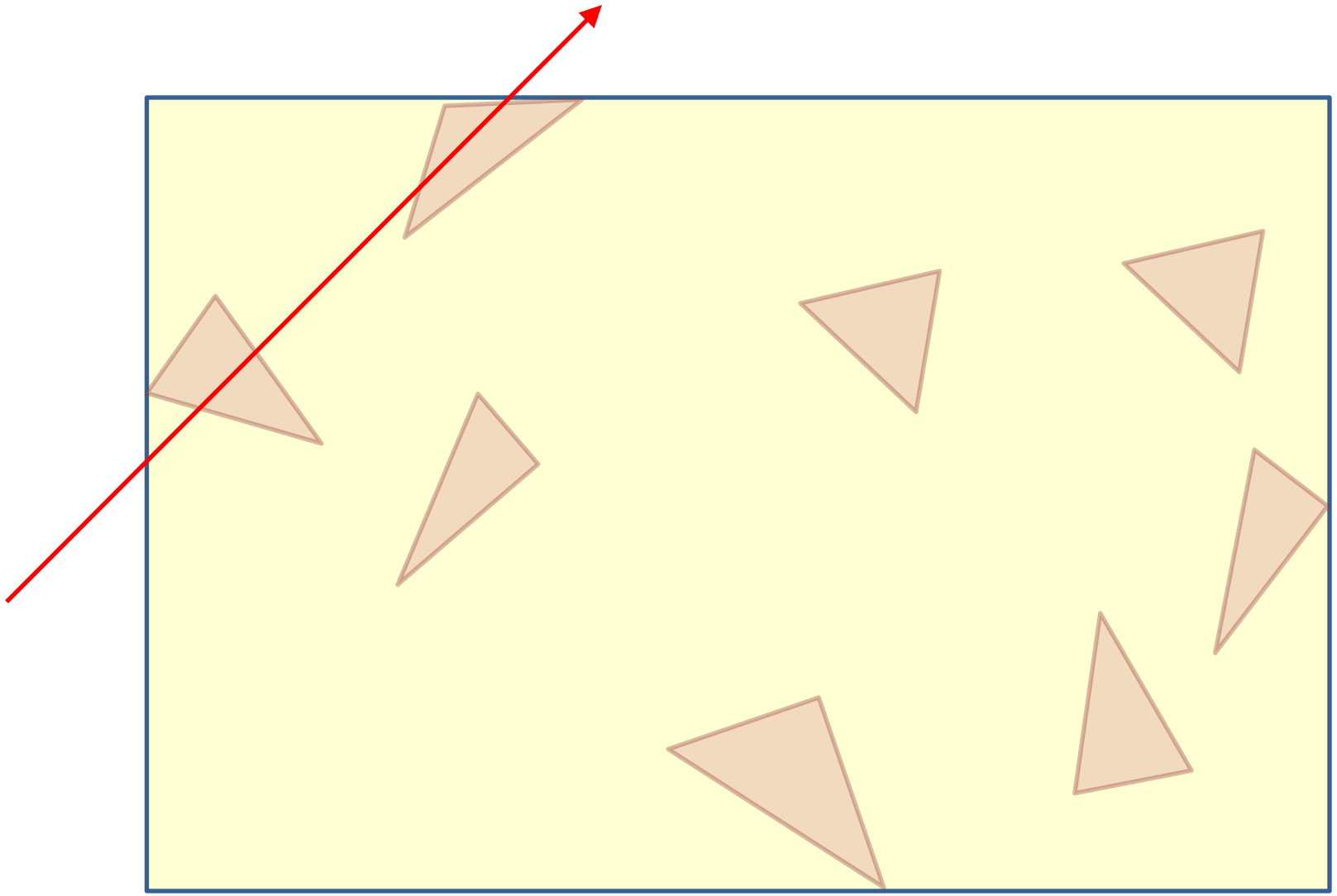
Ray Traversal

- Lets say that we've determined that the ray intersects the root volume (or even starts inside the root volume)
- We then need to test it against all of the child volumes (typically two) of the root node, and potentially recurse from there
- In order to make sure we traverse the children in order, the best approach is to first test the ray against all of the child volumes without recursion
- Then, determine which of the volumes gets hit first, and then recurse through the volumes in order
- When a collision is detected in a leaf node, we pop back up the recursion, but continue testing any volumes who were hit closer than the existing intersection
- If a closer hit is detected, we go with that one, and then continue until there are no closer volumes remaining
- This will make sure we finish testing all overlapping volumes before we stop searching- therefore assuring that we find the closest hit

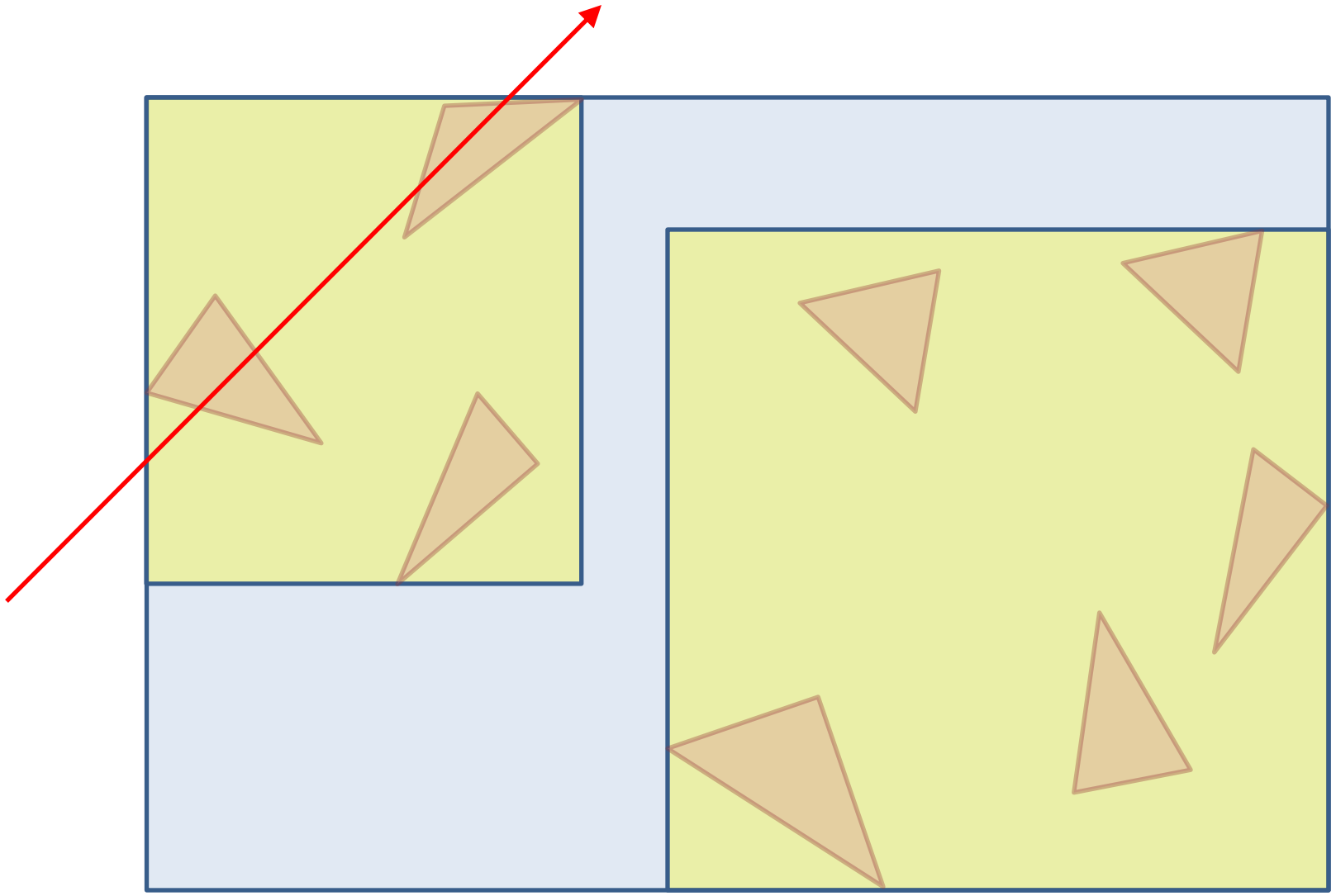
Ray vs. AABB-Tree



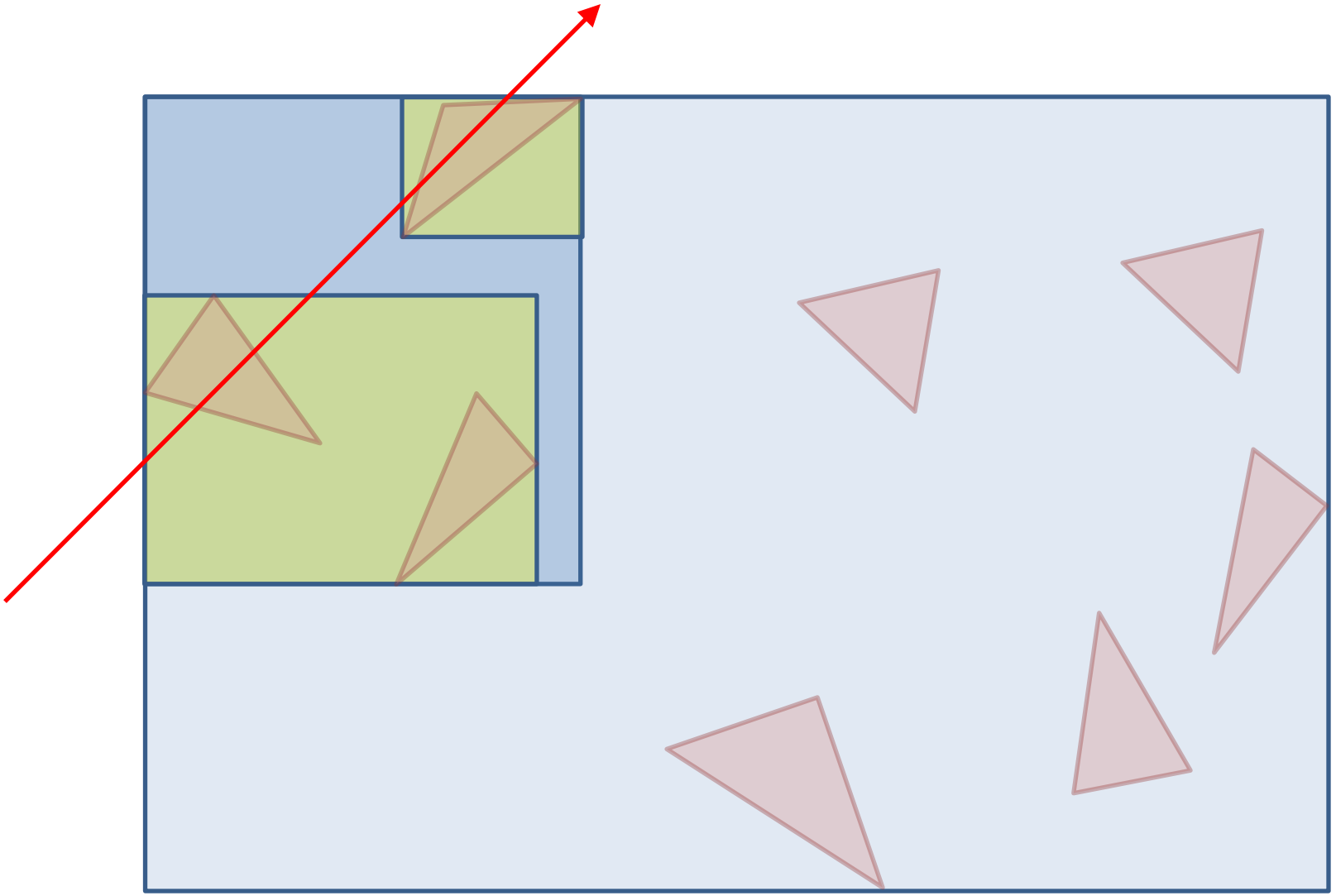
Ray vs. AABB-Tree



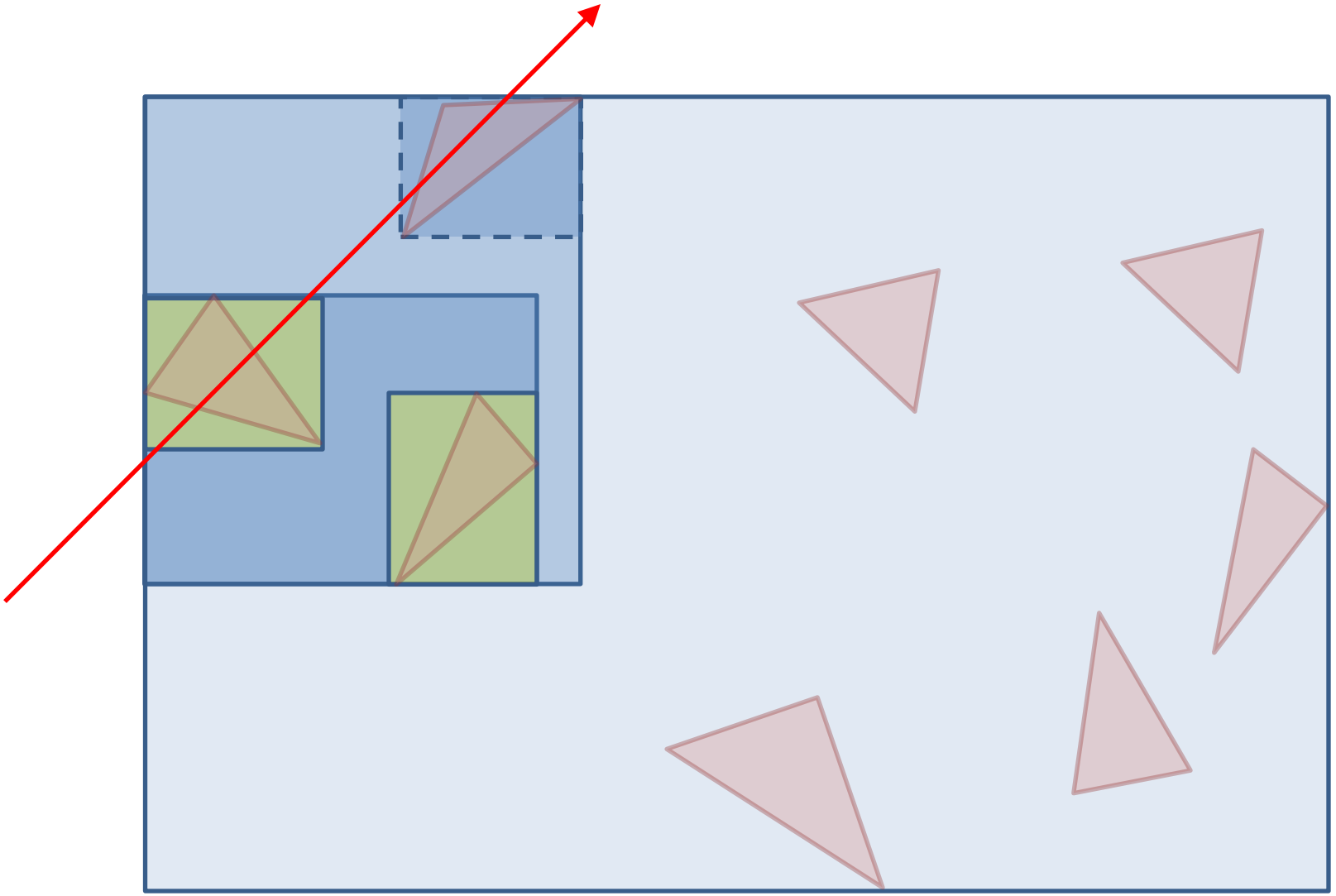
Ray vs. AABB-Tree



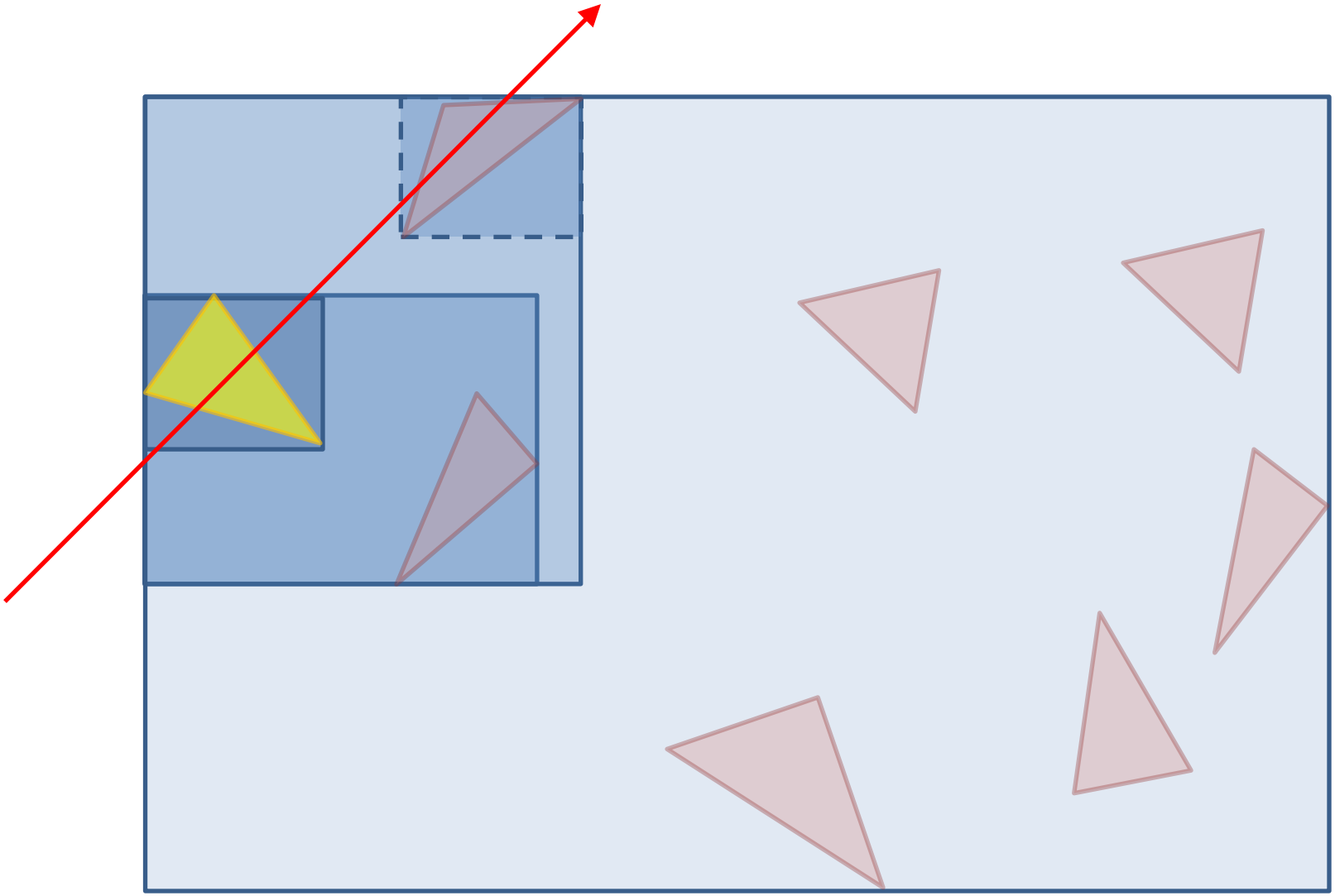
Ray vs. AABB-Tree



Ray vs. AABB-Tree



Ray vs. AABB-Tree



BVH Traversal Pseudocode

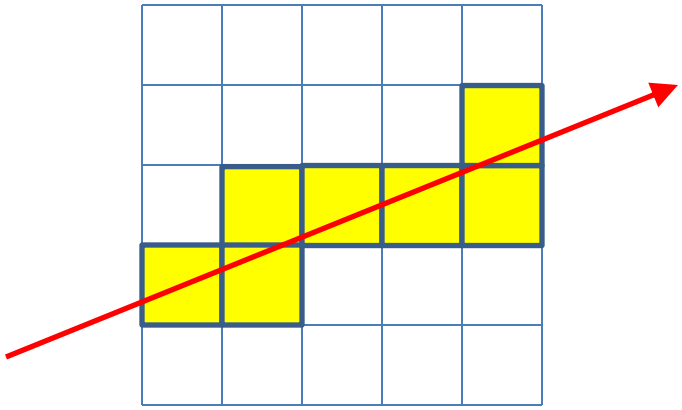
```
class BoundVolumeNode {
    bool IntersectVolume(const Ray &ray, Intersection &hit) {
        // Do geometric test with the volume itself and find the closest hit (if less than existing hit.HitDistance)
    }

    bool IntersectChildren(const Ray &ray, Intersection &hit) {
        // If this is a leaf node, test against triangles
        if(NumChildren==0) return IntersectTriangles(ray,hit);      // IntersectTriangles() just tests all triangles in the leaf

        // Test all child volumes
        Intersection volhit[NumChildren];
        for(each child) {
            volhit[child].HitDistance=hit.HitDistance;     // Avoid testing volumes closer than the best hit so far
            Child[child].IntersectVolume(ray,volhit[child]);
        }

        // Full recursive test on children, sorted in order of distance
        bool success=false;
        for(each child hit, sorted in order of closest volhit.HitDistance to farthest) {
            if(volhit[child].HitDistance < hit.HitDistance)
                if(Child[child].IntersectChildren(ray,hit)) success=true;
        }
        return success;
    }
};
```

Ray Traversal for Planar Partitions

- K-D trees and BSP trees are spatial partition data structures that use a single plane at each node to split the parent volume into two child volumes
- The ray traversal algorithm for planar partitions is structurally very similar to the general BVH algorithm with a couple exceptions:
  - Instead of testing intersections with the entire child volumes, planar partitions just need a simple test to determine which side the of the plane the ray origin is, and where on the plane the ray intersects
  - Also, being binary trees, there are only two possible orders to test the children, so there is no need for loops or explicit sorting, and a couple if() tests are all that is needed to implement the logic. The same is actually true for any binary BVH as well
  - Also, once we find an intersection in a child node, we are finished and don't need to test any other children
- This makes ray traversal against K-D (& BSP) trees very simple and fast, however there is a catch:
  - As some (or many) triangles will get split by planes, and so either need to placed redundantly into multiple leaf nodes, or be clipped into additional triangles
  - This adds some extra memory and performance overhead, but is balanced out by some of the other memory and performance benefits

Ray Traversal for Grid Structures

- Grid type data structures include uniform grids, nested grids, and octrees
- Ray traversal through the grid is analogous to the classic problem of rasterizing a line onto a grid (known as Bresenham's algorithm)



Scenes & Animations

Spatial Data Structures in Scene Graphs

- When rendering a scene, we could simply take all of our objects and build one big spatial data structure around everything
- This would allow a very simple architecture for the ray tracer as the entire scene would ultimately consist of a bunch of triangles in one big data structure. We would not need the Object base class for example…
- This is often fine for simple to medium complexity scenes
- However, for large scenes, we really want to be able to take advantage of instancing in most cases
- If we make a BoxTreeObject class as we described earlier, it should automatically be compatible with the InstanceObject class we described in an earlier lecture
- This means we can load an object and create a data structure for it once, and then instance it in the scene many times
- We can also take all non-instanced geometry and build one big data structure for that as well
- This approach can save a lot of memory, which is often important in large renders

Spatial Data Structures of Objects

- We can take it a step further if we want
- Instead of having our spatial data structures store Triangles at their leaf nodes, we can have them store Objects
- We can also derive Triangle itself off of Object
- In addition, we need to add a couple new virtual functions to the Object base class such as:

```
virtual bool ComputeInBox(const Vector3 &bmin,const Vector3 &bmax);
virtual void ComputeBoundBox(Vector3 &bmin,Vector3 &bmax);
```

- ComputeInBox() determines if the Object has any geometry inside the specified box, and ComputeBoundBox() computes an axis aligned box tightly fitting the Object
- This allows us to construct AABBs, Octrees, Nested Grids, and K-D Trees out of any set of Objects (including sets of mixed types of Objects)
- Therefore, we can build an initial AABB tree around a complex model, and then instance that model 100 times, and build a second AABB tree around the instances
- This allows a multi-level scene construction and provides an approach towards rendering huge scene complexity with lots of complex instanced models

Rendering Animations

- One nice thing about any spatial data structure is that it can be re-used across multiple frames when rendering an animation
- This amortizes the cost of creating the data structure, and allows us to spend more time creating the most optimal data structure to get the fastest per-frame render times
- One catch, however is when rendering deforming geometry like skin, cloth, or deformable solids
- It turns out that AABB trees are very well suited for these situations
- We can construct the tree once for the first frame, and then instead of rebuilding it entirely for each new frame, we can simply *update* it

AABB Trees and Deforming Geometry

- To update the AABB tree, we recurse from the top down to the leaf nodes
- Each leaf loops through its triangles and recomputes its own bounding box
- Then the recursion returns back up through the parent nodes
- Each parent recomputes their box simply by fitting around their child boxes
- This propagates back up to the root of the tree, which will ultimately result in a tight fitting box around the entire set of triangles
- This involves a fixed amount of computation per node in the tree, and because the total number of nodes in the tree is linearly proportional to the total number of triangles, the update operation is O(N) and very fast, requiring only simple floating point addition and subtraction
- One catch, however, is that if the object undergoes significant deformation, the tree structure itself will become suboptimal and require rebuilding from scratch. This isn't a big problem since the change from frame to frame is small
- Therefore, one can usually quickly update the AABB each frame and only rebuild periodically (like every 10 frames, or every time the cumulative volume of the data structure grows by some percentage)