Spatial Data Structures
Part 1

Steve Rotenberg

CSE168: Rendering Algorithms

UCSD, Spring 2014

Ray Intersections

- We can roughly estimate the time to render an image as being proportional to the number of ray-triangle intersection tests required
- This can be estimated as:

$$N_{total} = N_{pixels} * N_{rays} * N_{tris}$$

- Where

N_{total} = Total number of ray-triangle intersections

N_{pixels} = Numer of pixels (XRes * YRes)

N_{rays} = Average number of rays per pixel

N_{tris} = Average number of triangles tested per ray

Ray Intersections

$$N_{total} = N_{pixels} * N_{rays} * N_{tris}$$

- The number of pixels N_{pixels} is specified by the desired output format
- Performance in ray tracing therefore comes by minimizing the average number of rays per pixel N_{rays} and the number of triangles tested per ray N_{tris}
- Minimizing the number of rays per pixel N_{rays} (while maintaining the desired level of image quality) is the job of the shading system
- Minimizing the number of triangles per ray is the job of *spatial data structures* (or *acceleration structures*)

Render Performance

- Recall N_{tris} is the average number of triangles tested *per ray*
- Lets say that N_{scene} is the total number of triangles in the entire scene
- If every ray has to test every triangle, then the overall performance of the render would be *linear* with respect to the number of triangles in the scene
- In this situation, we would say that render performance is $O(N_{scene})$, or just O(N)

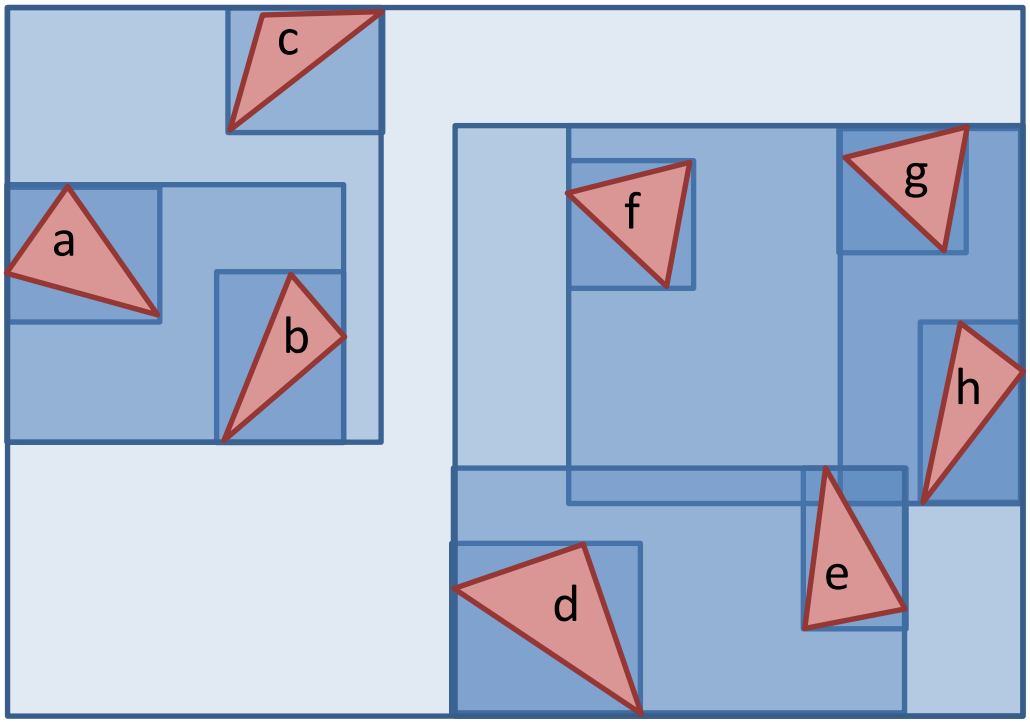
Render Performance

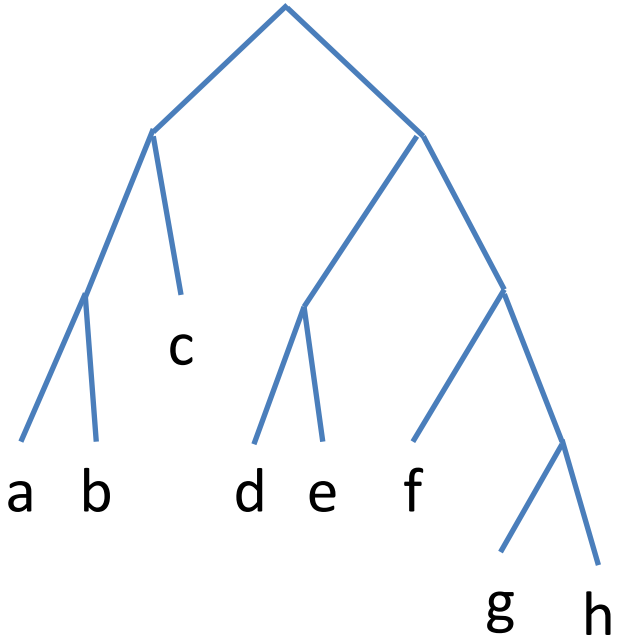
- Ordinarily, O(N) performance is considered pretty good in computer algorithms- and in many algorithms, is considered ideal
- However, for ray intersection, it turns out that the right spatial data structures can reduce this closer to *logarithmic* performance, or O(log N)
- This turns out to be really fantastic, as it means that multiplying the number of triangles in the scene by a fixed number has the effect of linearly increasing the render time
- This means that 1,000,000 triangles should be only a bit worse than 100,000 triangles. Or that a *billion* triangles is modestly slower than a *million*…
- This makes it possible to render scenes of vast geometric complexity
- If we consider that an HDTV 1080P image has 1920 x 1080 = > 2 million pixels, we should be able to render scenes with triangles smaller than pixels, which basically means we can have as much geometric detail as we want (or as much as we can fit in memory…)

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

Axis Aligned Bounding Box Tree





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
- This can be a time consuming process that itself often runs at O(N log N) performance
- O(N log N) is slower than linear, but we hope that the entire tree creation is still faster than the render itself. Ultimately however, as N grows, this can be the limiting factor to scene complexity
- Also, keep in mind that we often want to render animations, and in that case, we can build the tree once and re-use it across multiple animation frames
- Often, tree creation requires the use of *heuristics* to determine exactly how to split up the volumes

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

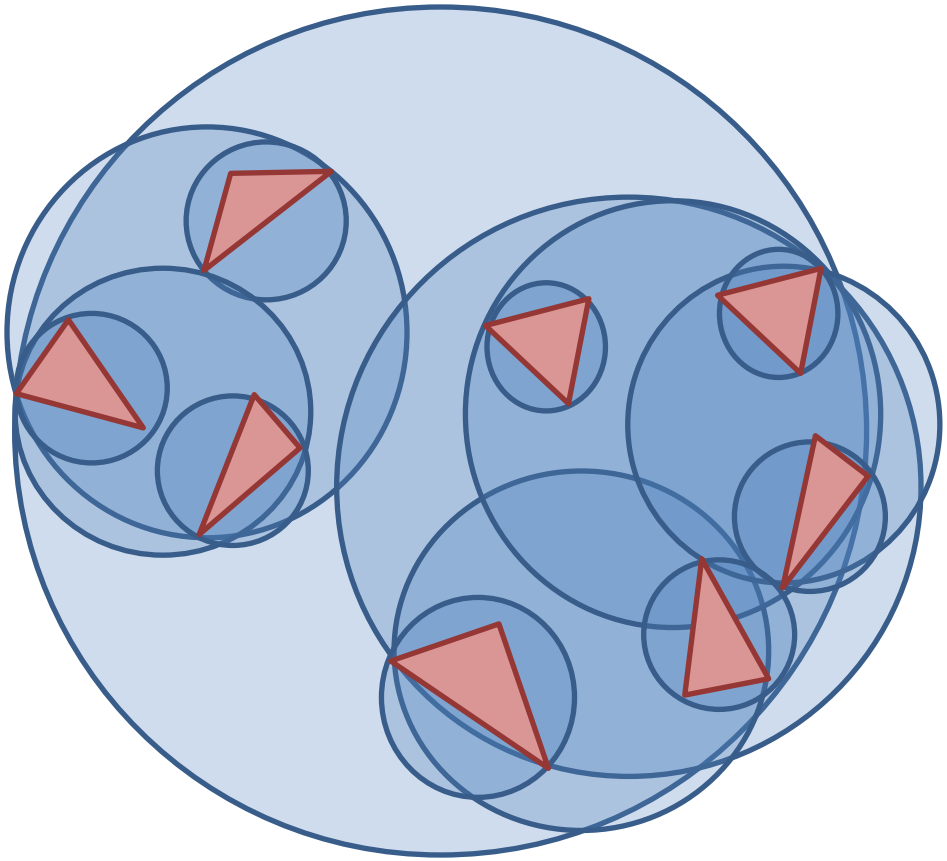
Hierarchical Data Structures

- There are various specific types of hierarchical data structures used in ray tracing
- As we mentioned, they all have similar construction and traversal functions
- The main difference is in how they represent the volumes used
- To classify the different structures, we will start by splitting them into two groups:
  - Bounding Volume Hierarchies (BVH)
  - Spatial Partitions

Bounding Volume Hierarchies

- BVHs use simple geometric volumes such as boxes or spheres at each level of the hierarchy
- Their main distinction from spatial partitions is that the volumes potentially can (and often do) overlap
- This isn't a big problem and in many ways actually makes things easier
- Common BVH types used in ray tracing include:
  - Spheres
  - Axis aligned bounding boxes (AABB)

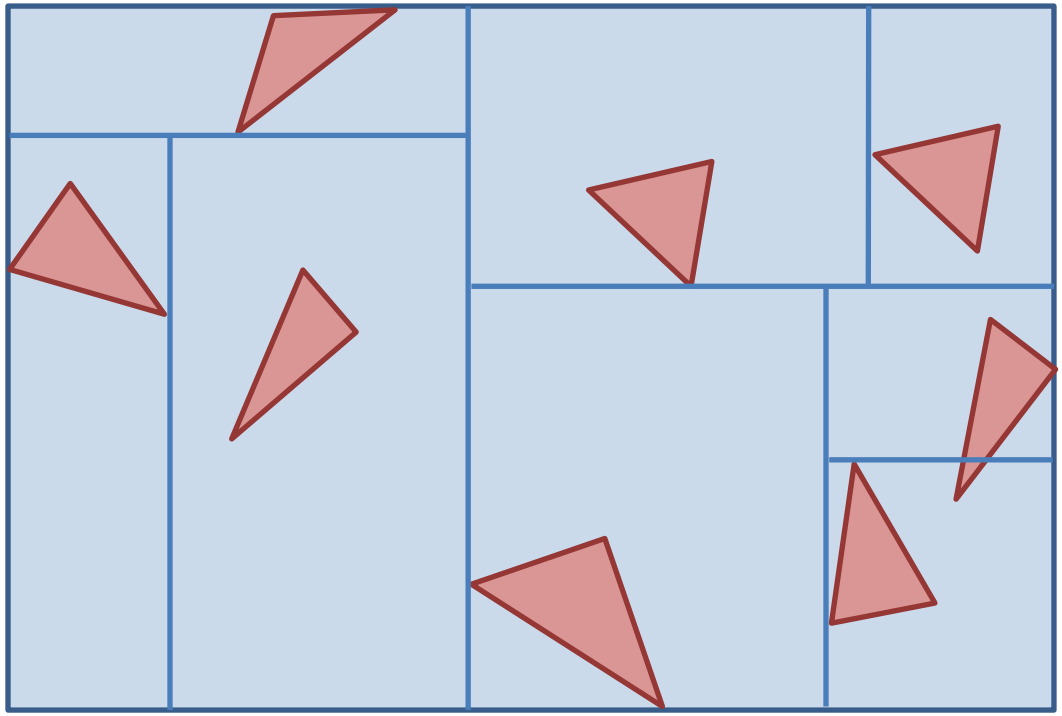
Sphere-Tree



Spatial Partitions

- Spatial partitions start with a top level enclosing volume (typically a box, but some can start with infinite space)
- At each level down in the tree, the volume is partitioned into non-overlapping sub-volumes
- Typically, this is done either with a single plane or with some sort of gridding
- For the single plane case, if the plane is aligned to the x, y, or z axis, then we have a *K-D tree*. If it is an arbitrary plane in space, then it is a *BSP tree* (*binary space partition*). Both of these techniques can start with a box at the top level or with infinite space at the top level
- For the gridding case, if we start with a cube and split each level into 2x2x2 sub-cubes, then we have an *octree*. If we split each level into a NxNxN (where N is typically 10 or less), then we have a *nested grid*
- Spatial partitions have to deal with the fact that triangles may either end up in multiple volumes or will need to be clipped along volume boundaries. This is one of the difficulties that BVHs avoid by allowing volumes to overlap

K-D Tree



Non-Hierarchical Data Structures

- Almost all data structures used in rendering of complex scenes are hierarchical in nature
- However, for some simpler cases, or for some special purpose situations (such as GPU rendering), it is occasionally useful to have non-hierarchical data structures
- One possible example is a uniform grid
- One advantage of grids (as with nested grids) is that ray traversal through the grid cells is very quick, as it equates to a 3D equivalent of the classic 2D line rendering algorithm (Bresenham's algorithm)

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

Spatial Data Structures

- Overall, sphere trees are nice for simple renders, but are rarely used in large scale commercial renderers or in complex scenes
- AABB trees are simple to build, relatively simple to traverse, plus have an additional advantage of working very well with animating geometry
- BSP trees can be tricky to construct because it is difficult to choose an arbitrary plane to partition the geometry effectively. These were popular in the past but have not seen as much use lately
- K-D trees are easier to construct than BSP trees and tend to be a fairly popular and general purpose choice
- Nested grids are also popular, particularly for tessellated models or other models with lots of small, similarly sized triangles
- Octrees are really just a special case of nested grids but are a bit easier to implement
- Many other spatial structures have been tested and used in the past

Axis Aligned Box Trees

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 around 10 or less

Binary vs. Non-Binary Trees

- Some tree types are always binary (K-D, BSP)
- Some others are always non-binary (nested grid, octree)
- BVH's (sphere trees, AABB trees) however could be binary or not
- Making them binary keeps the implementation simple for both tree construction and ray traversal
- However, with some more coding effort, it is conceivable that N-trees could outperform in certain situations if the tree construction could take advantage of clustering in the scene geometry

Leaf Nodes

- How many triangles should we store per leaf node?
- If we limit the leaves to one triangle, then we end up with more levels of the tree and more time spent in tree traversal
- If we allow a small number of triangles, we trade off the last couple levels of tree traversal for a few more triangle intersection tests
- The best optimal choice depends on the relative cost of testing a triangle to testing a box
- Typically, some experimentation is needed to find an optimal value and this may vary based on camera angle or scene layout, so a reasonable compromise has to be chosen

BoxTreeNode Class

```
class BoxTreeNode {
public:
        BoxTreeNode();
        ~BoxTreeNode();
        bool Intersect(const Ray &ray, Intersection &hit);
        void Construct(int count, Triangle **tri);
private:
        Vector3 BoxMin, BoxMax;
        BoxTreeNode *Child1, *Child2;
        Triangle *Tri[MaxTrianglesPerBox];
};
```

NOTE: Uses a fixed sized array of MaxTrianglesPerBox to reduce allocation overhead and improve cache coherence. If MaxTrianglesPerBox is large, then it would probably be better to allocate an array or even use a std::vector

BoxTreeObject Class

```
class BoxTreeObject:public Object {
public:
        BoxTreeObject();
        ~BoxTreeObject();
        void Construct(MeshObject &mesh);
        bool Intersect(const Ray &ray, Intersection &hit);

private:
        BoxTreeNode *RootNode;
};
```

Tree Construction

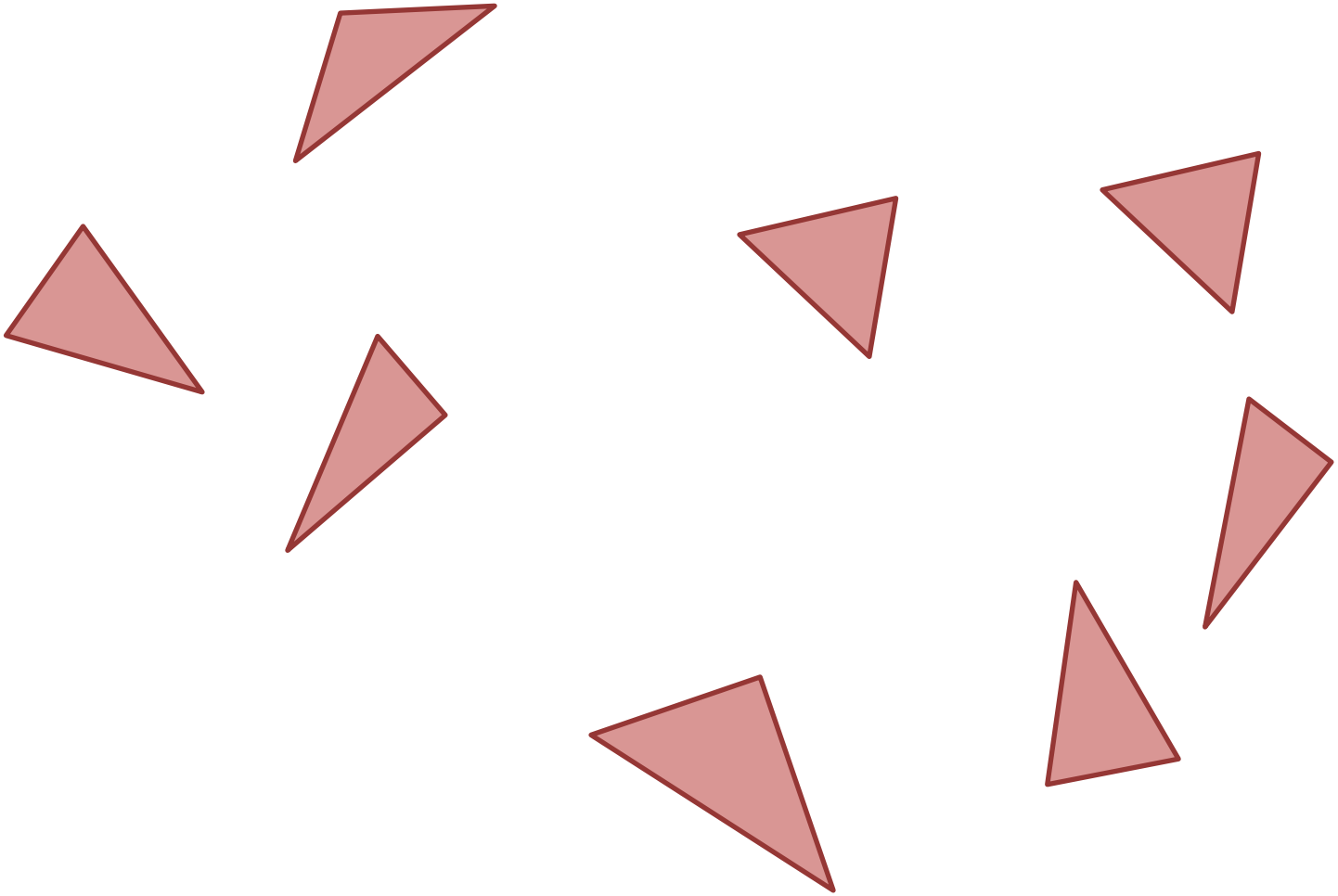
Tree Construction

- There are several approaches to constructing an AABB tree (and most of these apply to any type of hierarchical data structure)
- The two main ways to approach the problem are either *top-down* or *bottom-up*

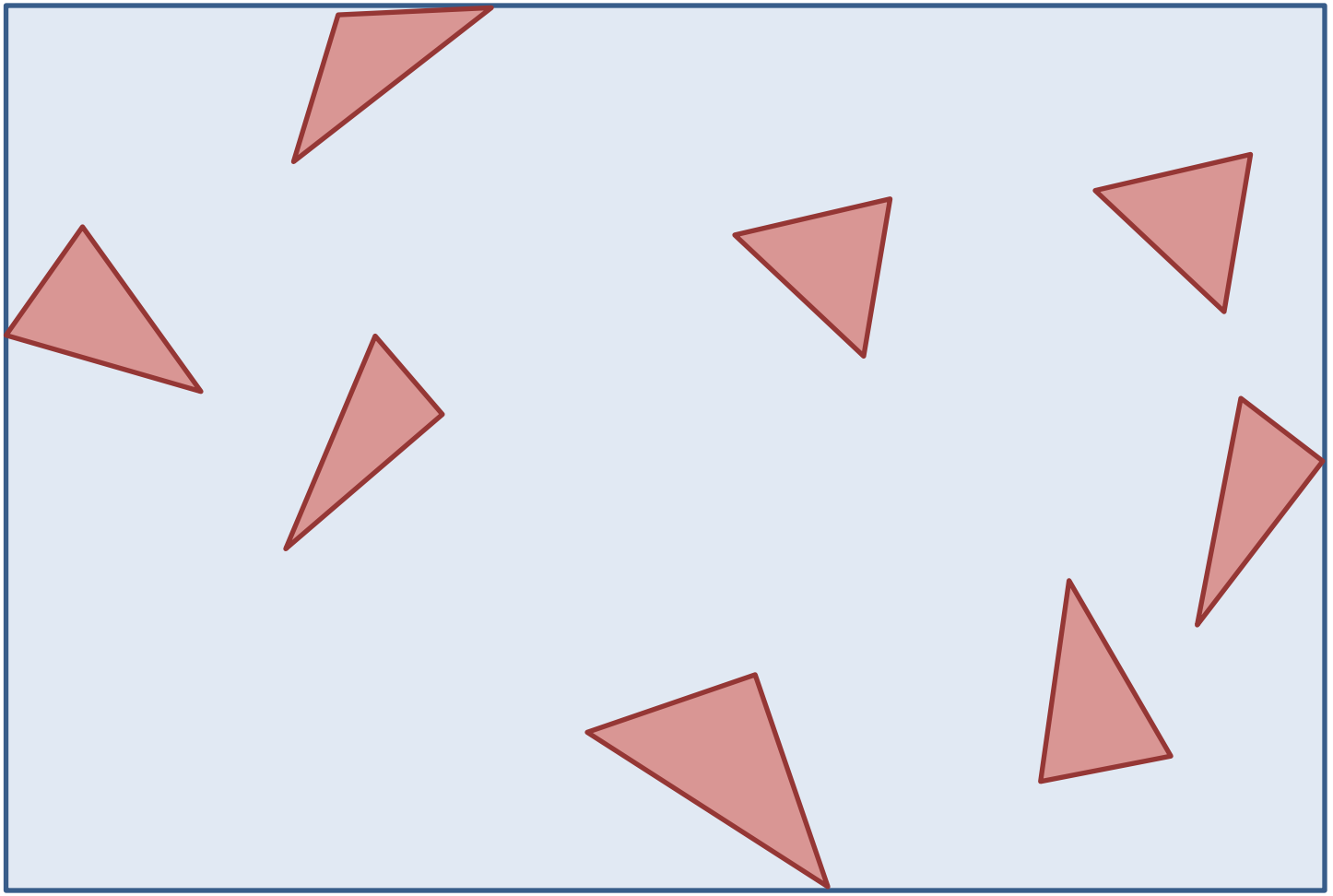
Top-Down Construction

- Top-down construction starts by looking at the entire array of input triangles and building the top level node as the box that tightly encompasses all of the triangles
- It then proceeds by determining a way to split the triangles into two (or possibly more) groups
- Splitting is typically done by choosing an axis aligned plane and then putting each triangle into a group according to which side of the plane the triangle falls into (note that some triangles may intersect the plane, so we choose based on the center of the triangle)
- Once we have two groups, we create a child box for each one and repeat the process with the subgroups
- This is recursively repeated until the group size is smaller than the maximum number of triangles per leaf node

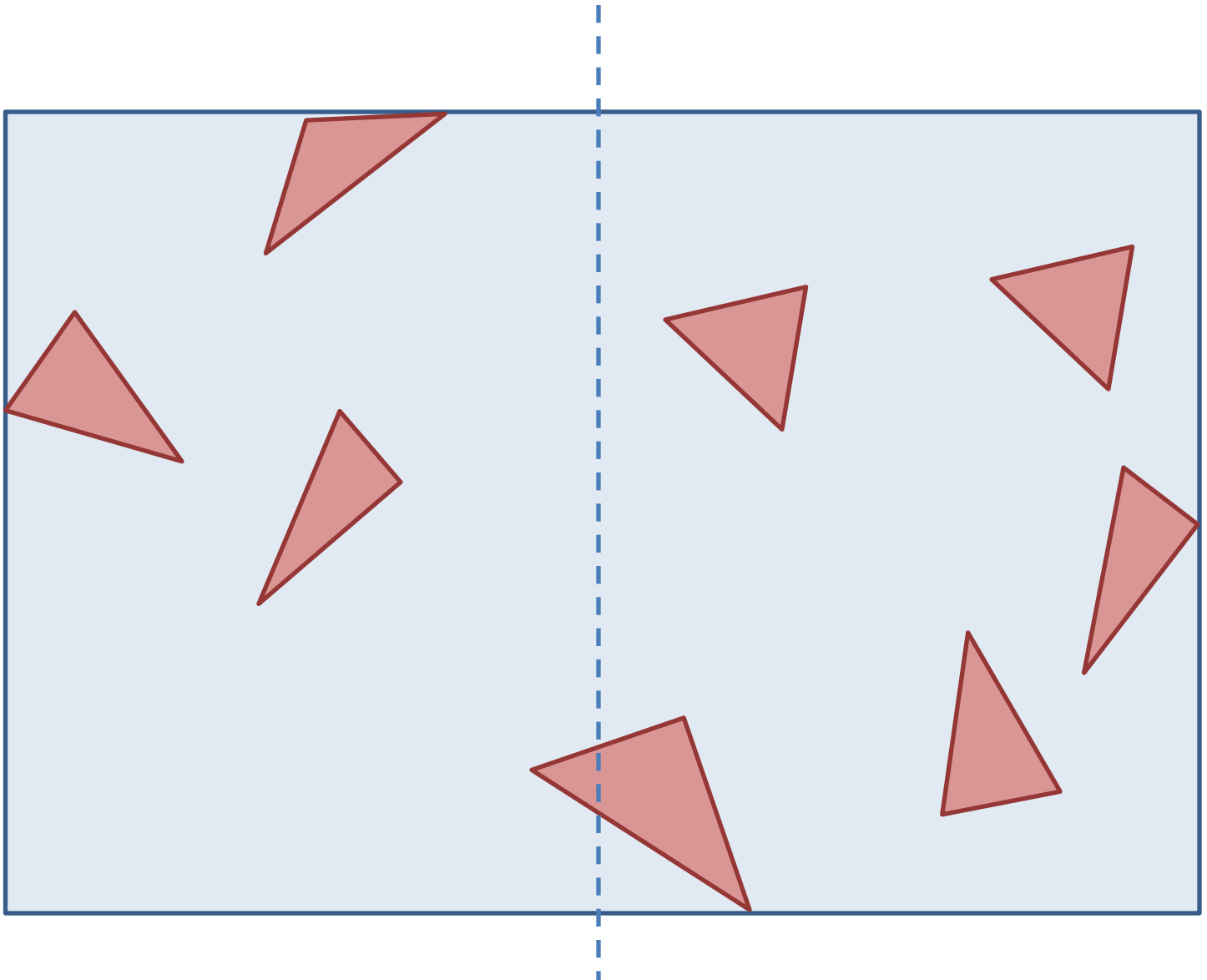
AABB-Tree



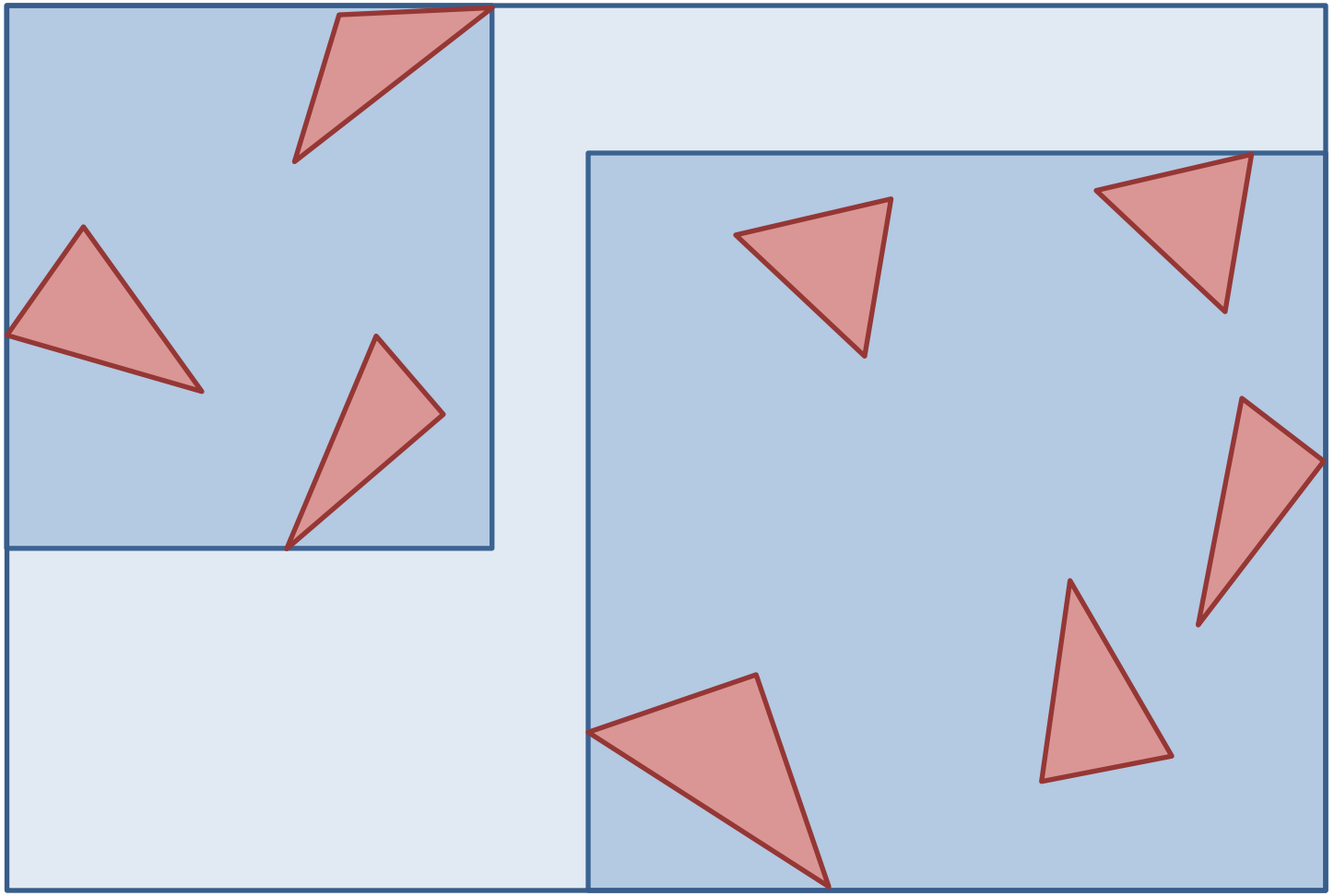
AABB-Tree



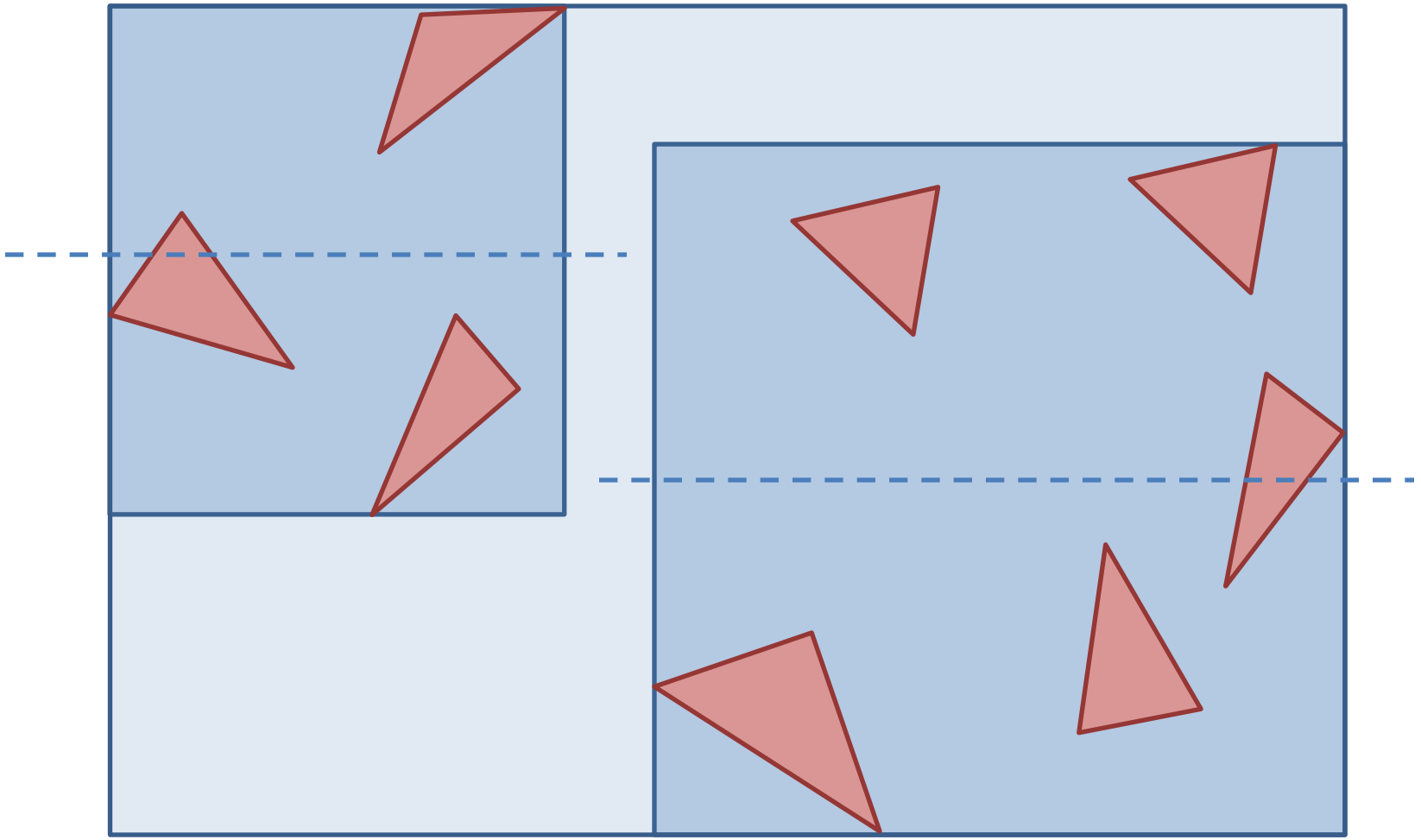
AABB-Tree



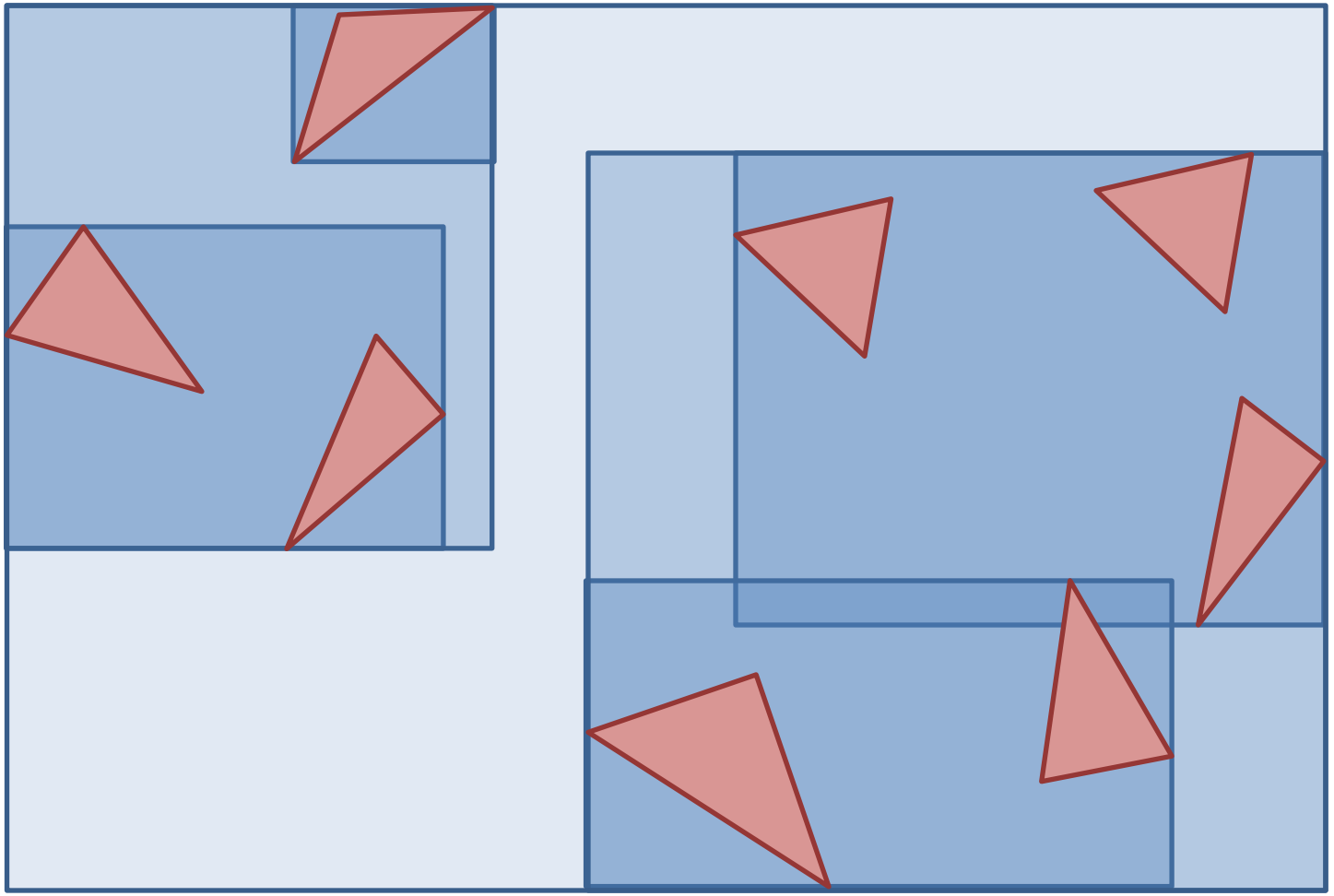
AABB-Tree



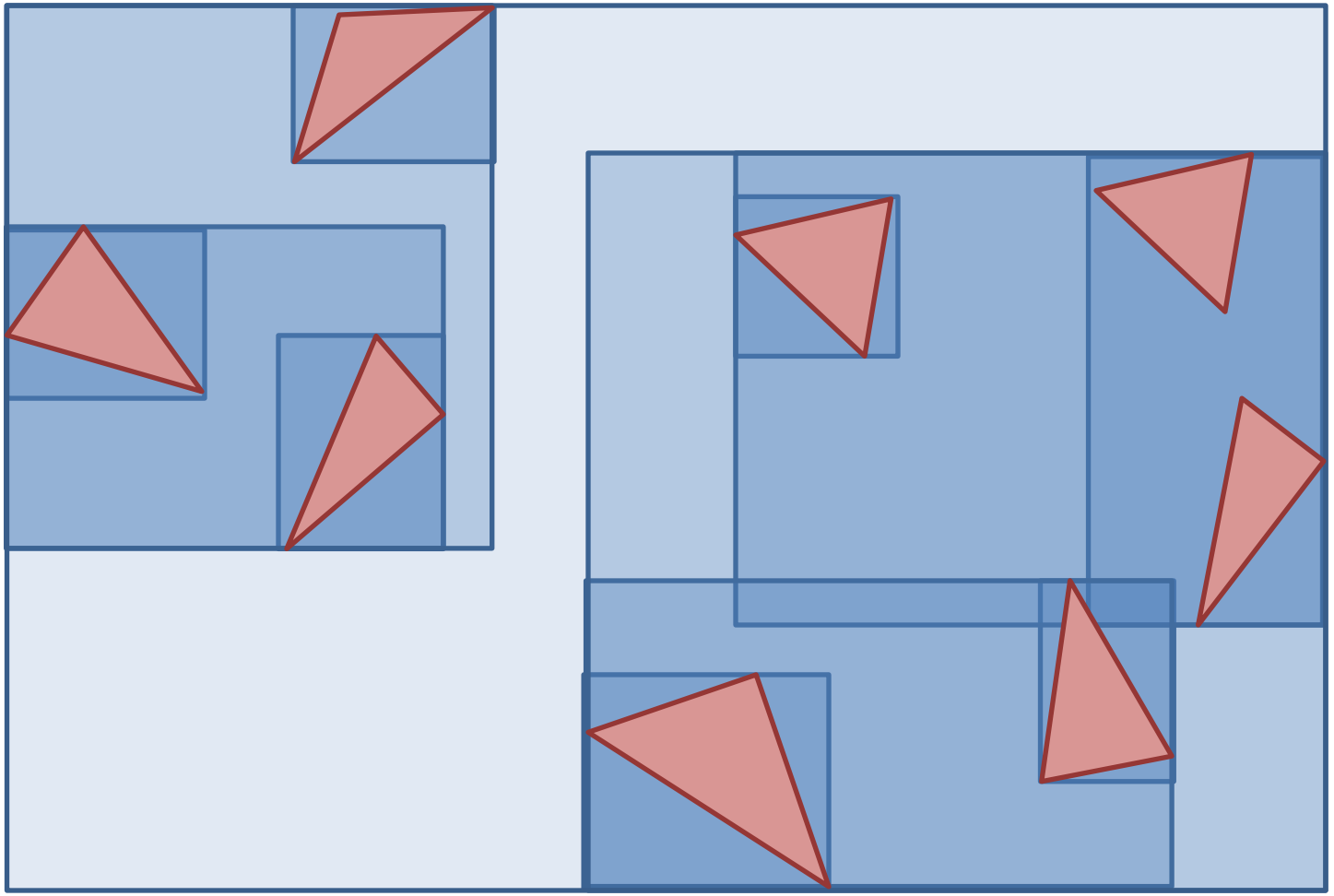
AABB-Tree



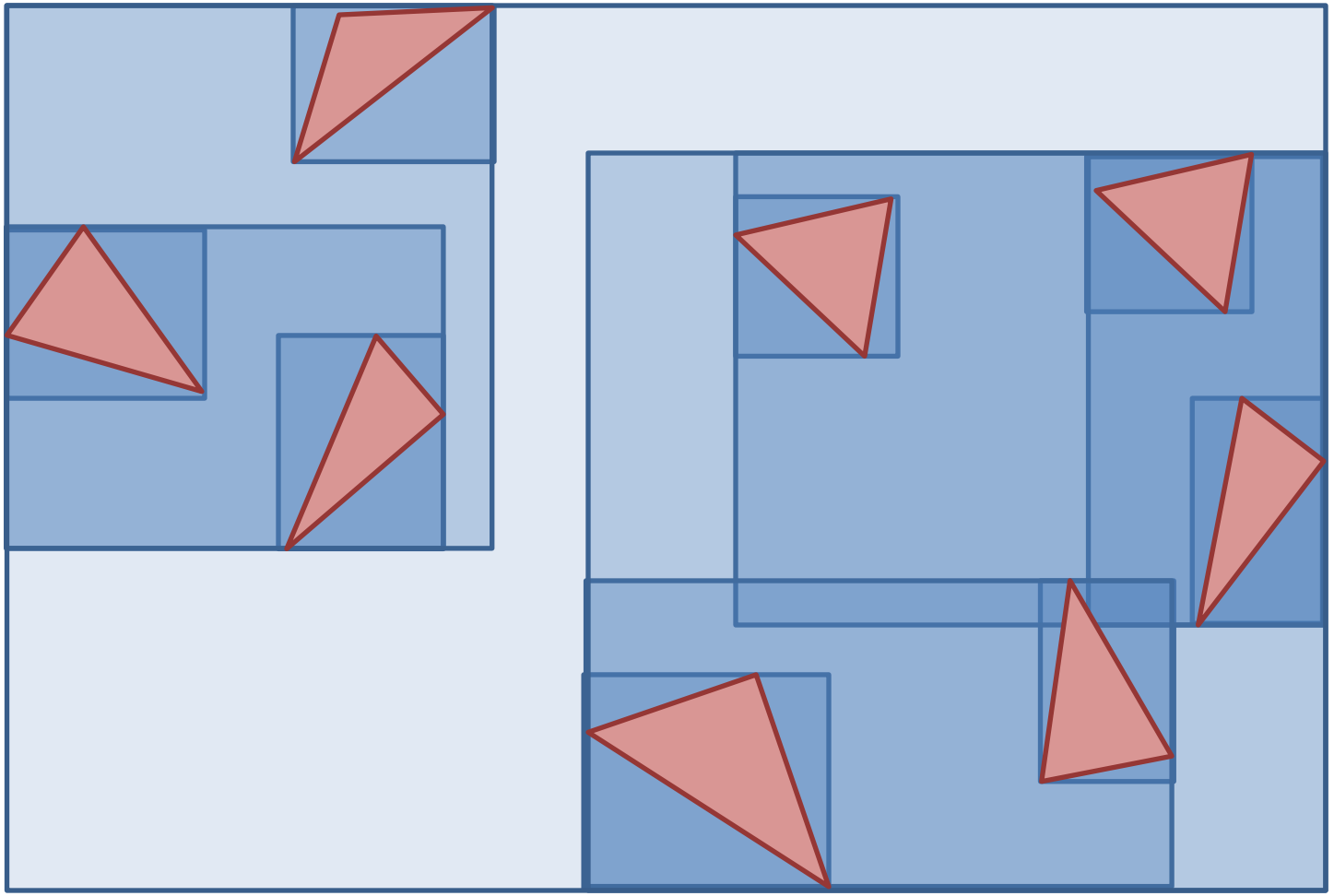
AABB-Tree



AABB-Tree



AABB-Tree



Top-Down Issues

- One potential problem that arises is that after finding the splitting plane and putting the triangles into two groups, one of the groups may have all the triangles and the other may be empty
- In practice, this only happens when the number of triangles in the initial group is pretty small, or in situations where there are lots of overlapping triangles
- One simple fix is just to move any one triangle into the empty group and proceed
- Another option is to switch to a different algorithm for splitting the groups, or just use some other heuristic

Top-Down Algorithm

```
void BoxTreeNode::Construct(int count,Triangle **tri) {
        // Compute BoxMin & BoxMax to fit around all tri's

        // Check if this is a leaf node
        if(count <= MaxTrianglesPerBox) {
                // Copy triangles to BoxTreeNode's Tri array
                return;
        }

        // Determine largest box dimension x, y, or z
        // Compute splitting plane halfway along largest dimension

        // Allocate two new temporary arrays
        Triangle *tri1=new Triangle*[count];    Triangle *tri2=new Triangle*[count];
        int count1=0, count2=0;

        // Place triangles into group 1 or group 2
        for(each tri) {
                // Compute center of triangle & determine which side of splitting plane
                // Add to appropriate group
        }

        // Check if either group is empty. If so, move (at least) 1 triangle into that group

        // Recursively build sub-trees
        Child1=new BoxTreeNode;         Child2=new BoxTreeNode;
        Child1->Construct(count1,tri1);     Child2->Construct(count2,tri2);

        // Free up arrays
        delete []tri1;      delete []tri2;
}
```

Top-Down Algorithm

- The algorithm on the previous slide was set up to take an array of Triangle pointers as input
- It could be easily extended to handle Object pointers by adding a virtual function to the Object base class that computes a bounding box for the Object
- Triangle could also be derived off of the Object base class, allowing a very general purpose data structure that could contain triangles, instances, other box-trees, spheres, and more

Splitting Heuristics

- How we divide a group of triangles up into two groups can have a big impact the performance of the ray traversal as well as the performance of the construction algorithm itself
- Many different *splitting heuristics* have been proposed to address this issue
- Perhaps the simplest box splitting heuristic is to select the longest axis of the box and then split halfway down that axis
- This might work OK to get started, but there are other options that are can be useful

Area Heuristic

- One popular heuristic is called the *area heuristic*
- It attempts to minimize the product of the area of each box with the number of primitives in the box
- At each level, it considers several possible splitting planes and selects the one that minimizes:

$$a_1 n_1 + a_2 n_2$$

where a_1 is the area of child box 1 and n_1 is the number of primitives contained in box 1, etc.

Area Heuristic Performance

- The algorithm tests several (9 recommended) splitting planes regularly spaced across the x, y, and z axes and chooses the best one
- This results in a increase in construction time linearly proportional to the number of planes tested, so 9 in x, y, and z would cost 27 times as long to generate the data structure as just selecting the midpoint of the largest axis
- This sounds very bad, but if it reduces the render time by minutes, it may be worth the extra seconds to do this…
- Also note that the area heuristic results in significant performance benefits for scenes with irregular and clustered distributions of triangles, such as scenes made up of various high detailed objects spaced around
- For single objects of uniformly high detail (like the dragon), it actually makes very little difference in render performance

Bottom-Up Construction

- Bottom-up construction begins by grouping nearby triangles into small clusters, and then the leaf nodes are created to enclose the clusters
- Those clusters are then *agglomerated* into larger clusters, and on and on until we get one large cluster for the root of the tree

Clustering Algorithms

- Both top-down and bottom-up methods can benefit from the large body of research in *clustering algorithms (*or *cluster analysis*)
- These are algorithms that are typically used to partition a large set of points into a set of reasonable clusters according to some geometric relationship
- These are used for a lot of data analysis applications and in computer vision, but don't often find application in computer graphics
- They are typically designed for points, but can be adapted to cluster geometric objects such as triangles
- Common clustering algorithms include k-means, expectation-maximization (EM), hierarchical clustering…

Grid Construction

- Gridded data structures (such as uniform grids, nested grids, and octrees) are constructed in a similar process, but generally not requiring any heuristics
- For an octree, for example, one starts by computing the bounding box of the entire triangle set, and then set the root node to be a uniform cube fitting around that box
- If more than N_{max} triangles are in the box, it is split at the center into 2x2x2 boxes and the triangles are placed into the appropriate boxes. This step is repeated recursively until there are below N_{max} triangles in a box
- Note that when splitting the triangles into child groups, some triangles may end up into more than one group. These can either be clipped along the box boundaries (creating more triangles) or we can allow triangles to exist in multiple leaf nodes in the tree

Grid Construction Issues

- One situation to watch out for is cases where many triangles (more than N_{max}) come together to a point, such as at the tip of a cone
- No amount of recursive subdivision will reach a leaf node with fewer than N_{max} triangles
- In this case, we have to put an upper limit on the recursion depth and allow special case nodes to contain more than N_{max} triangles
- This can cause similar trouble with other data structures like K-D trees, but less so with BSP trees
- Also, BVH's don't have this problem, but can suffer from sub-optimal performance for rays that come near these areas

Ray Traversal

- We will discuss ray traversal of spatial data structures in the next lecture

Project 2

- For Project 2, you will implement a hierarchical data structure (such as an AABB tree) and render scenes with many triangles
- Also, add shadow ray testing for each light source
- In addition, add some internal timers to test how long it takes to create the data structure and how long it takes to render

Sample Program

```
main() {
    Scene scene;

    // Sun light
    DirectLight sun;
    sun.SetDirection(Vector3(2.0, 1.0, 1.0));
    sun.SetColor(Color(1.0, 1.0, 0.8));
    scene.AddLight(sun);

    // Load mesh
    MeshObject mesh;
    mesh.Load("dragon.ply");

    // Create box tree
    BoxTreeObject boxtree;
    boxtree.Construct(mesh);

    // Create instance 1
    InstanceObject instance1;
    instance1.SetChild(boxtree);
    instance1.GetMatrix().MakeRotateX(0.5);
    instance1.GetMatrix().d.Set(-1.0, 1.0, 0.0);
    scene.AddObject(instance1);

    // Create instance 2
    InstanceObject instance2;
    instance2.SetChild(boxtree);
    instance2.GetMatrix().MakeRotateY(-1.0);
    instance2.GetMatrix().d.Set(1.0, 1.0, 0.0);
    scene.AddObject(instance2);

    // Create ground mesh
    MeshObject ground;
    ground.MakeBox(5.0, 0.05, 5.0);
    scene.AddObject(ground);

    // Render
    Camera camera;
    camera.SetFOV(40.0);
    camera.SetAspect(1.33);
    camera.SetResolution(800, 600);
    camera.LookAt(Vector3(1, 0.5, 2),Vector3(0, 0.5, 0));
    camera.Render(scene);
    camera.SaveBitmap("project2.bmp");
}
```

Project 2