

Lecture #16

Ray Tracing – Acceleration Structures

Computer Graphics
Winter Term 2020/21

Marc Stamminger / Roberto Grosso

Introduction

- Billions of rays are required to generate high quality images
- Two major components have to be optimized
 - Basic primitive tests, e.g. ray-triangle intersection
→ Lecture #15
 - strategies and data structures to reduce the number of necessary tests
→ this lecture

Ray Tracing Complexity

- Plain Ray Casting:

```
for each of the  $m$  pixels  
    test eye ray against each of the  $n$  scene primitives
```

- $O(mn)$ is enormous!
- typically $m, n > 1.000.000$
- 1 mio objects, 1 mio pixels \rightarrow 1 trillion (10^{12}) intersection tests...
- Ray Tracing: > 1.000 secondary rays per pixel common
 $\rightarrow 10^{15}$ intersection tests
- But: with acceleration structures the inner loop can be accelerated to $O(\log n)$
 \rightarrow entire algorithm becomes $O(m \log n) \rightarrow$ practical!

Ray Tracing – Acceleration Techniques

- Usually, 90% of the time goes into intersection tests
- Strategies for speeding up ray tracing:
 - Restrict intersection tests to objects close to ray
 - vice versa: quickly reject large groups of objects that cannot intersect
- Requires a preprocessing step to group objects
- Preprocessing should be lightweight (e.g. $O(n)$ or $O(n \log n)$) so that it amortizes

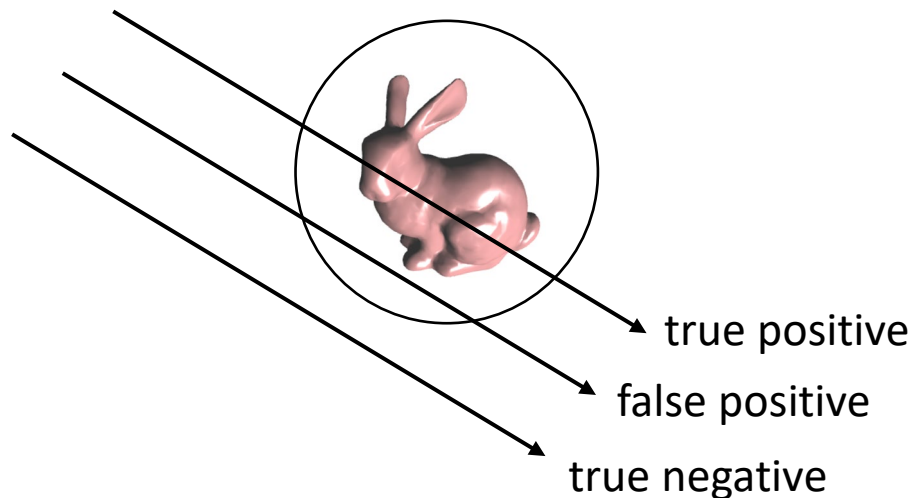
Acceleration Techniques

- Basis for most acceleration structures:

Bounding volumes (BV):

Find (geometrically simple) surrounding volumes for the complex objects

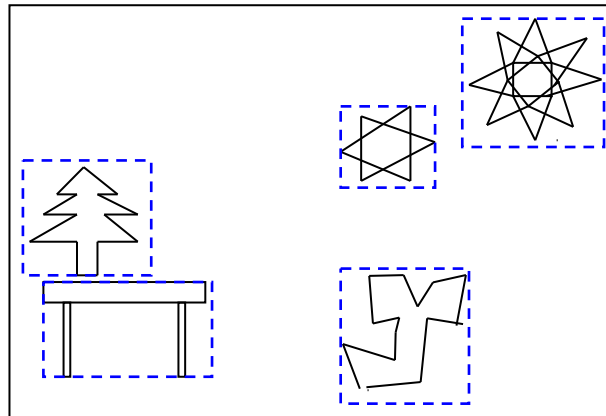
- Spheres
 - bounding boxes
- Before testing all inside objects, check for intersection with BV
→ if BV is not hit, fast trivial reject of all children
- Choose BV such that the intersection test is simple and efficient
 - spheres
 - axis-aligned boxes
 - ...



Acceleration Techniques

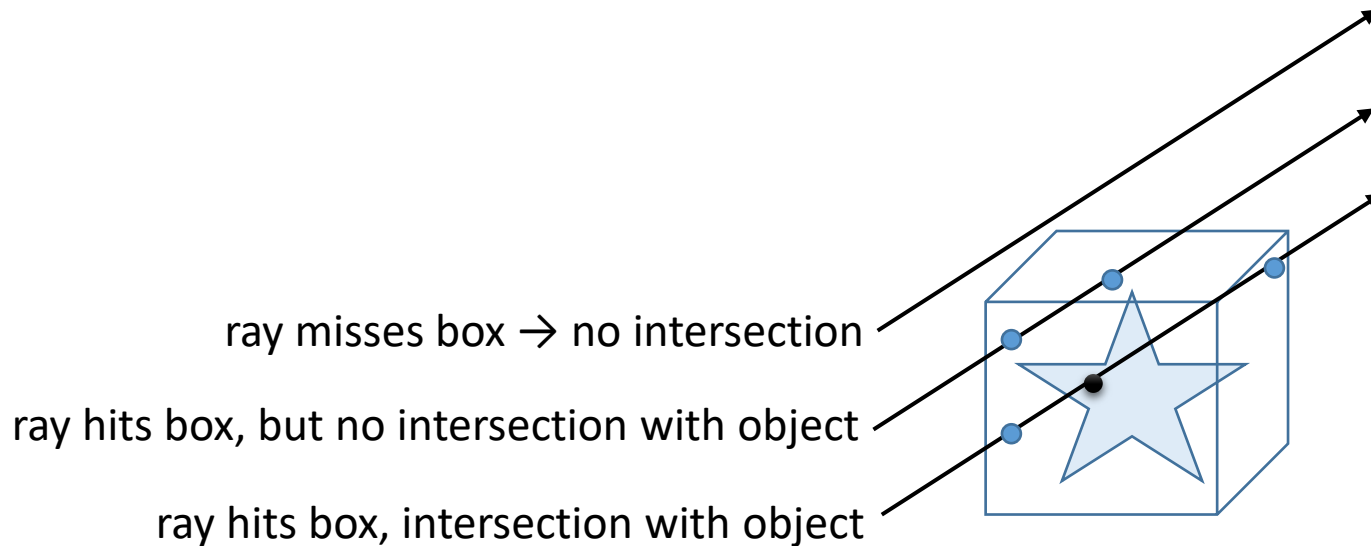
- Bounding volumes (BV) – Intersection test

```
if (intersect (ray, BV) == true) then  
    intersect(ray, BV.objects);  
end if
```



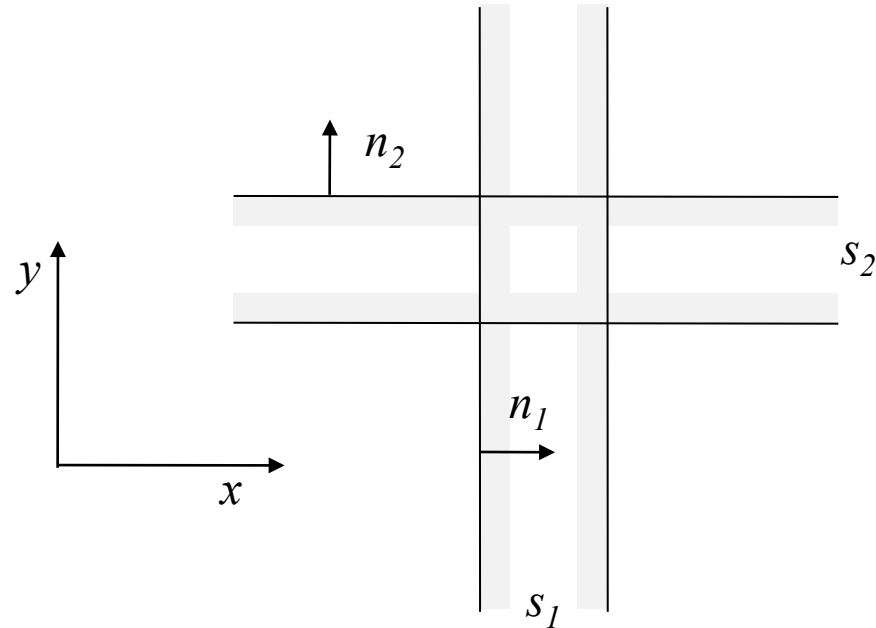
Ray – Box Intersection

- A typical BV is an axis aligned box
 - axes of box are aligned to x , y , and z -axis
 - box defined by minimal and maximal x , y , and z -coordinates
- Simple acceleration:
 - compute a box surrounding a complex object
 - if ray misses this bounding box, no tests with complex object necessary



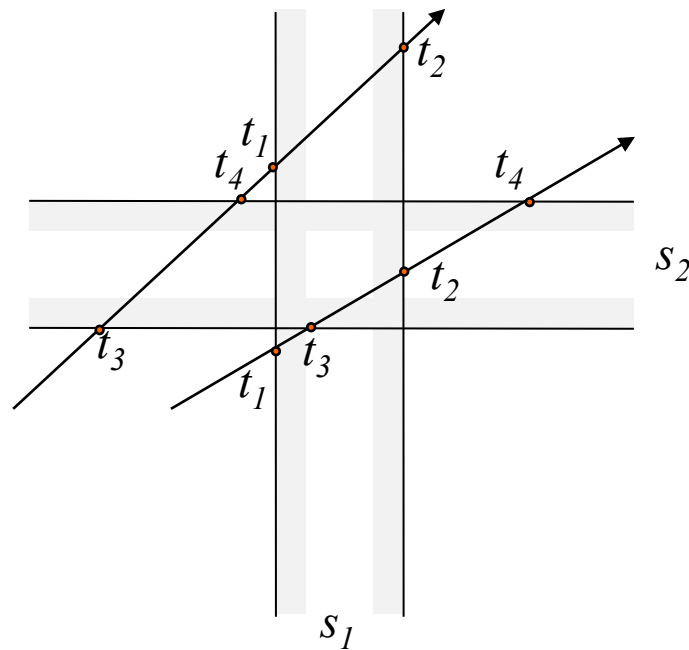
Ray – AABB Intersection

- AABB: axis aligned bounding box
 - box is aligned with the main axes
 - Intersection of three **slabs** $[x_{min}, x_{max}]$, $[y_{min}, y_{max}]$, $[z_{min}, z_{max}]$



Ray – AABB Intersection

- Intersection test
 - In 3D a point is inside the AABB if and only if it is inside all the three slabs
 - a ray intersects the AABB if and only if the intersection segments of the ray with the three slabs are overlapping

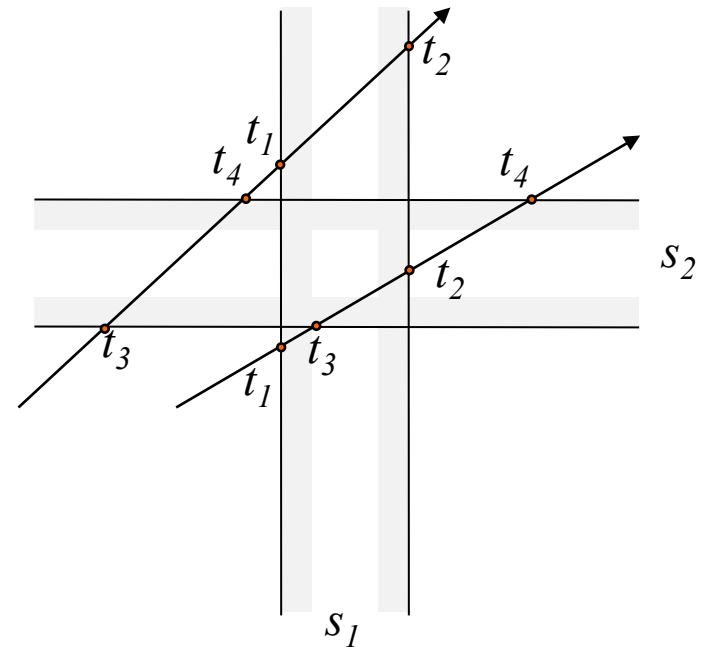


Ray – AABB Intersection

- Intersection test with ray

$$p(t) = e + td$$
$$t_n = \frac{d_n - e \circ n}{d \circ n}$$
$$t_f = \frac{d_f - e \circ n}{d \circ n}$$

- Intersection with s_1 is $[t_1, t_2]$
Intersection with s_2 is $[t_3, t_4]$
→ ray intersects iff $[t_1, t_2] \cap [t_3, t_4] \neq \{\}$



Ray – AABB Intersection

$$t \in [t_{xmin}, t_{xmax}]$$



$$t \in [t_{ymin}, t_{ymax}]$$

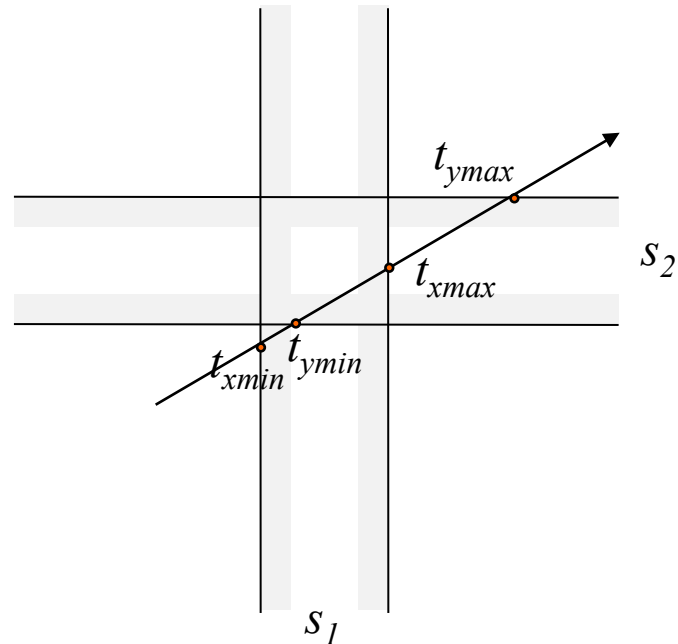


$$t \in [t_{xmin}, t_{xmax}] \cap [t_{ymin}, t_{ymax}]$$





$$t_{entry} = \max\{t_{xmin}, t_{ymin}\}$$

$$t_{exit} = \min\{t_{xmax}, t_{ymax}\}$$



Ray – AABB Intersection

$$t \in [t_{xmin}, t_{xmax}]$$


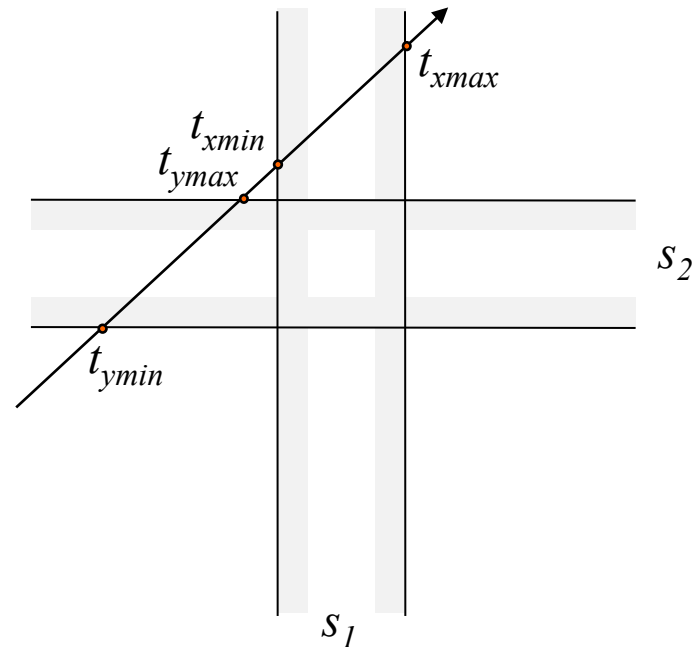
$$t \in [t_{ymin}, t_{ymax}]$$


$$t \in [t_{xmin}, t_{xmax}] \cap [t_{ymin}, t_{ymax}]$$


$$t_{entry} = \max\{t_{xmin}, t_{ymin}\}$$

$$t_{exit} = \min\{t_{xmax}, t_{ymax}\}$$

$$t_{entry} > t_{exit} \rightarrow \text{no intersection !}$$



Ray – AABB Intersection

- Representation of an AABB

```
// region R = { (x, y, z) | min.x<=x<=max.x,  
// min.y<=y<=max.y, min.z<=z<=max.z }  
struct AABB {  
    Point min;  
    Point max;  
};
```

Ray – AABB Intersection

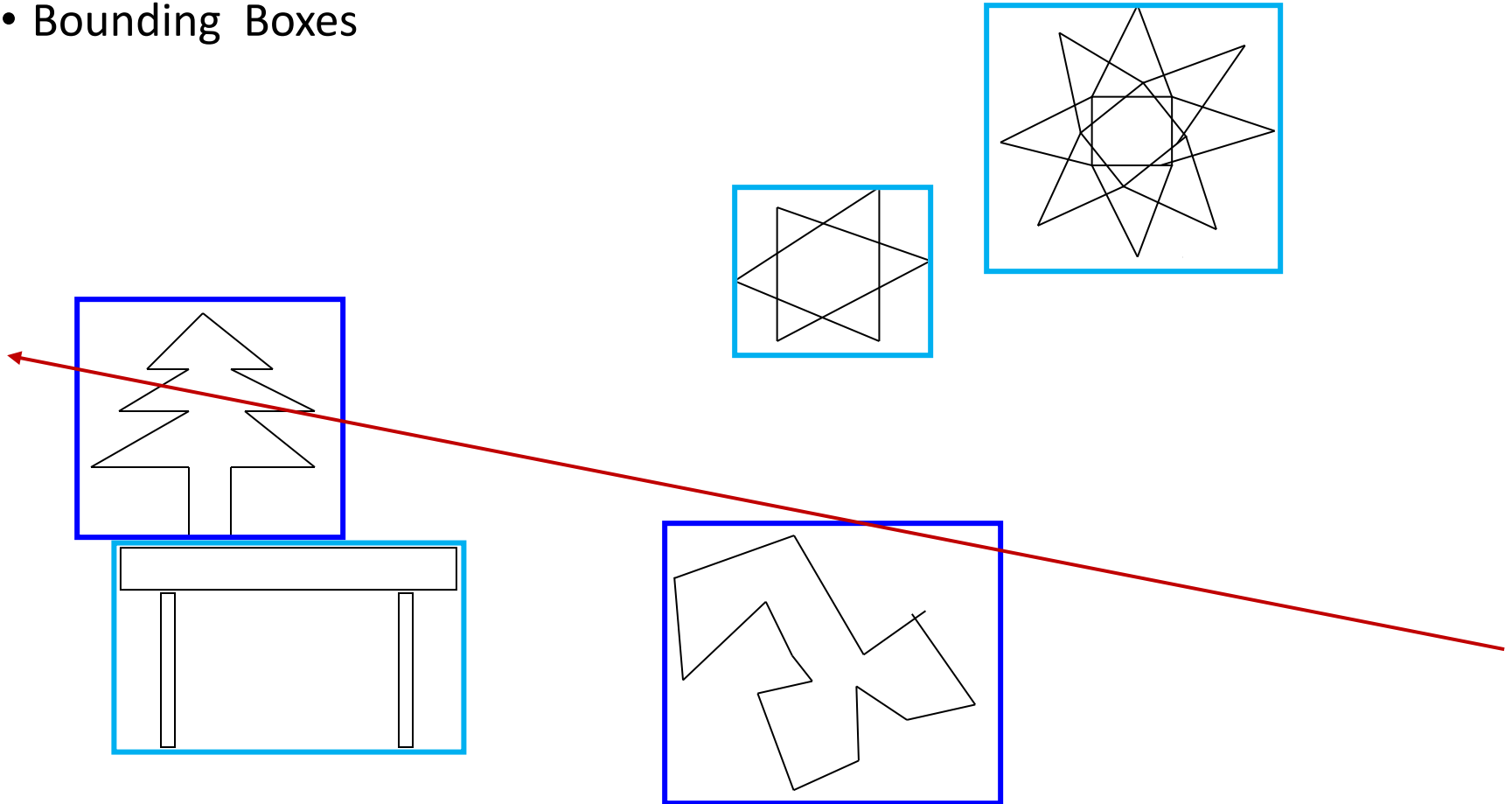
```
// Intersect ray  $R(t) = p + t \cdot d$  against AABB a. When intersecting,  
// return intersection distance tmin and point q of intersection  
int IntersectRayAABB(Point p, Vector d, AABB a, float &tmin, Point &q)  
{  
    tmin = 0.0f;           // set to -FLT_MAX to get first hit on line  
    float tmax = FLT_MAX; // set to max distance ray can travel (for segment)  
  
    // For all three slabs  
    for (int i = 0; i < 3; i++) {  
        if (Abs(d[i]) < EPSILON) {  
            // Ray is parallel to slab. No hit if origin not within slab  
            if (p[i] < a.min[i] || p[i] > a.max[i]) return 0;  
        } else {  
            // Compute intersection t value of ray with near and far plane of slab  
            float ood = 1.0f / d[i];  
            float t1 = (a.min[i] - p[i]) * ood;  
            float t2 = (a.max[i] - p[i]) * ood;  
            // Make t1 be intersection with near plane, t2 with far plane  
            if (t1 > t2) Swap(t1, t2);  
            // Compute the intersection of slab intersections intervals  
            tmin = Max(tmin, t1);  
            tmax = Min(tmax, t2);  
            // Exit with no collision as soon as slab intersection becomes empty  
            if (tmin > tmax) return 0;  
        }  
    }  
  
    // Ray intersects all 3 slabs. Return point (q) and intersection t value (tmin)  
    q = p + d * tmin;  
    return 1;  
}
```

Ray – AABB Intersection

- The test is a special case of the intersection test of ray against a Kay-Kajiya slab volume, T. Kay and J. Kajiya, SIGGRAPH 1986
- The Kay-Kajiya test is a specialization of the Cyrus-Beck clipping algorithms, M. Cyrus and J. Beck, Computer and Graphics, 1978

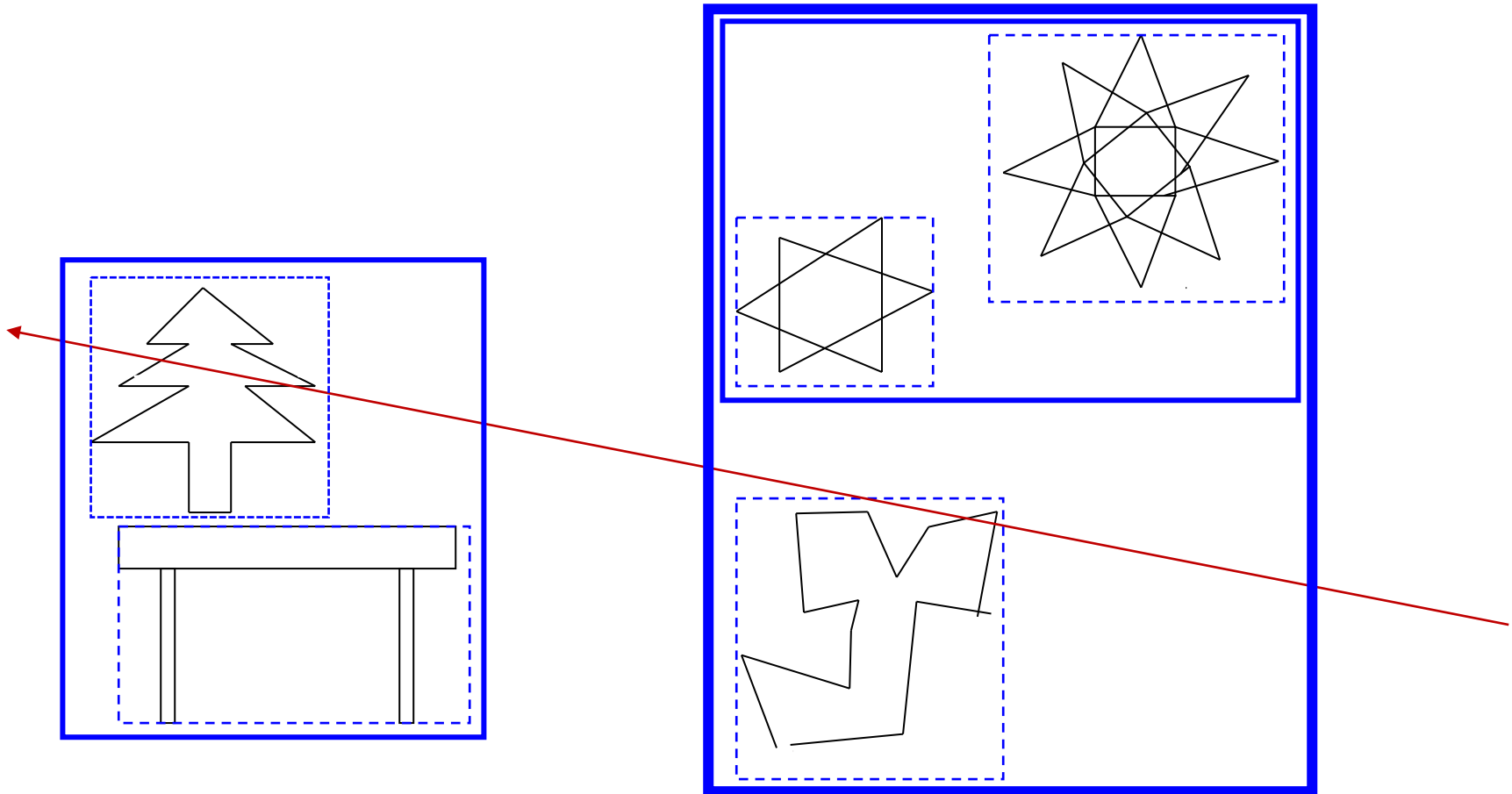
Acceleration Techniques

- Bounding Boxes



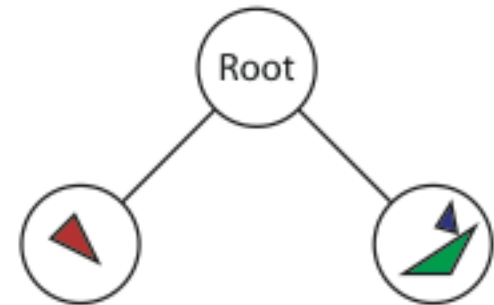
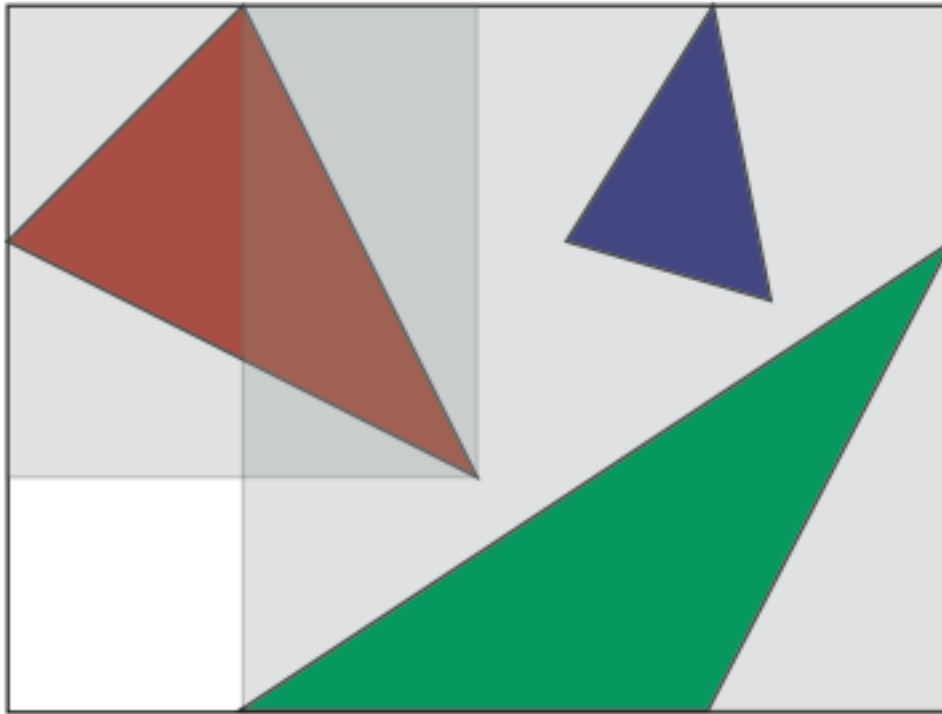
Acceleration Techniques

- Bounding Box Hierarchies



Acceleration Techniques

- BVH: Bounding Volume Hierarchy



Bounding Volume Hierarchy

- Construction
 - Recursive construction algorithm
 - Initialize root node. Fill it with all triangles in the scene
 - Recursively subdivide the root node
 - Stop recursion if depth of node exceeds a given value or the node contains less than a given number of triangles
 - Split the node into a left and right child otherwise

Bounding Volume Hierarchy

- Construction

```
buildTree(Scene& scene) {  
    create root node containing all triangles;  
    subdivide(root);  
}  
  
subdivide(Node& node) {  
    if (node.depth == MAX_DEPTH ||  
        node.numTriangles <= MIN_TRIANGLES)  
        return;  
    compute optimal split position;  
    create left and right child node;  
    sort triangles into left and right node;  
    compute bounds of left and right node;  
    subdivide(left);  
    subdivide(right);  
}
```

Bounding Volume Hierarchy

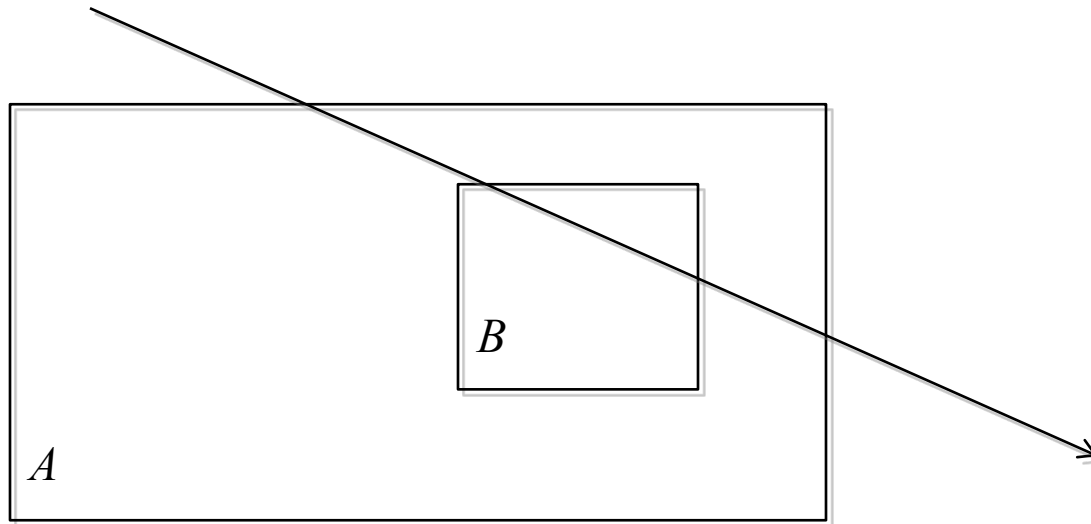
- Computing the optimal splitting plane
 - spatial median: split nodes in the middle along axis with largest extent
 - object median: split nodes so that left and right children contain the same number of triangles
 - cost function: minimize a cost function

Bounding Volume Hierarchy

- Surface Area Heuristics, SAH

Theorem:

- given a box B completely contained in a box A
- the probability that a ray traversing A intersects B is given by $SA(B)/SA(A)$, $SA(\cdot)$ is the surface area.



Bounding Volume Hierarchy

- Cost function for a split

$$C = C_T + |P_l| \frac{SA(B_l)}{SA(B_p)} + |P_r| \frac{SA(B_r)}{SA(B_p)}$$

- where P_l and P_r are the number of primitives in the left and right child nodes respectively, and B_l , B_r and B_p are the left, right and parent bounding boxes. C_T is the cost of computing a ray-primitive intersection relative to the cost of traversing a node.

Bounding Volume Hierarchy

- Problem statement
 - find the split position which minimizes the cost function
- Computing the SAH
 - Difficult task, the search space is extremely large!
→ all possible partitions → impossible for e.g. 1mio triangles
- [Ingo Wald: “On fast Construction of SAH-based Bounding Volume Hierarchies”, 2007:](#)

We consider a set of possible “splitting planes” in x, y, and z direction

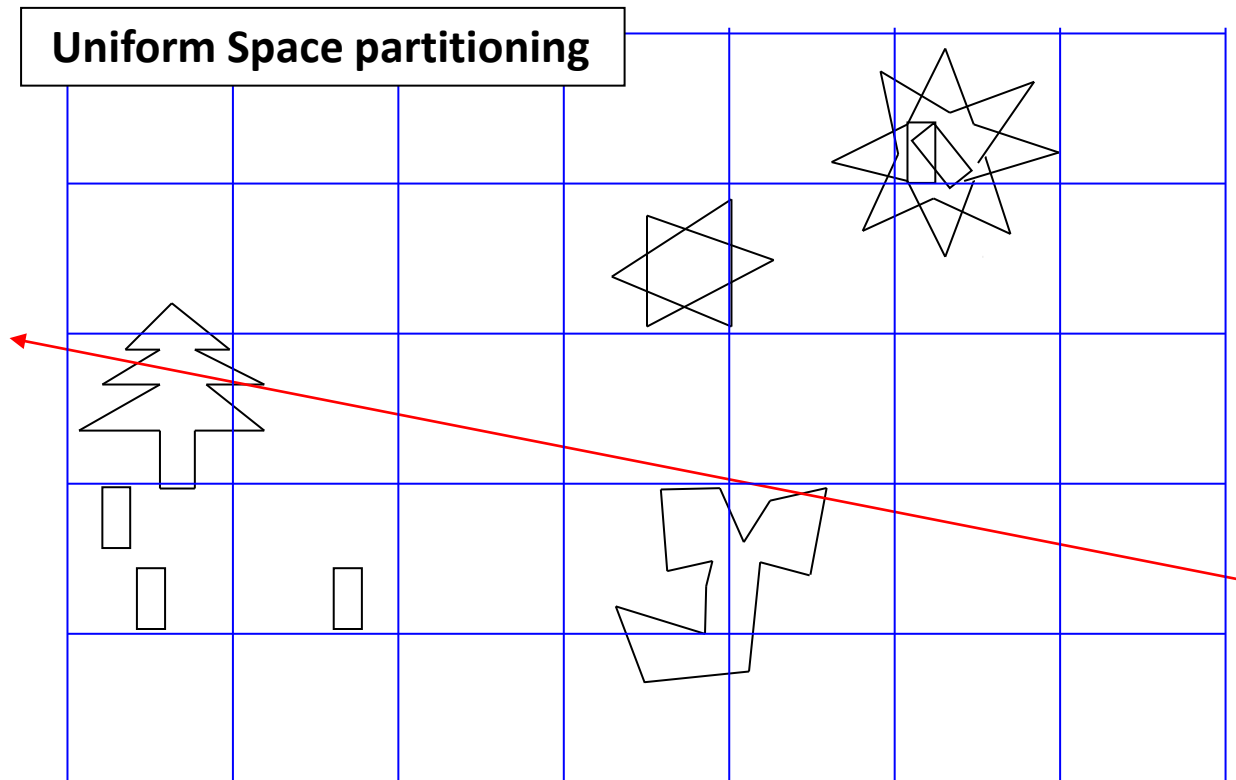
- compute bounds in x, y, and z
- generate split planes in constant distances in each dimension
- split BV along these planes
→ triangles intersecting the plane have to be sorted into one of the children
- compute BV for the two children for each plane
- use split plane, for which SAH predicts lowest cost

Bounding Volume Hierarchy

- adapts well to arbitrary geometries
- memory consumption is predictable
- each geometric primitive (e.g. triangle) occurs in exactly one leaf node
- nodes can overlap in space
- recursive traversal algorithm (use a stack!)

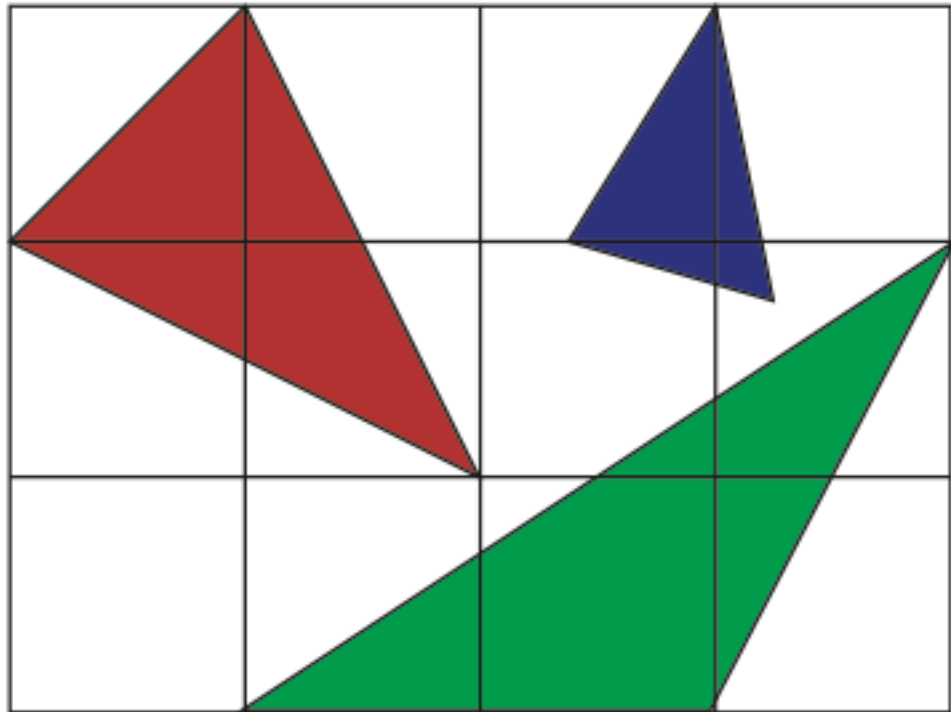
Spatial Hierarchies

- BVHs are built based on the objects → **object hierarchy**
- Alternatively, one can also build a hierarchy in space → **spatial hierarchies**
- Simplest version: uniform grid



Spatial Hierarchies – Uniform Grid

- Uniform Grid
 - store for each cell intersecting triangles
 - for ray intersection test: traverse the grid cells along ray
 - check triangles inside cells

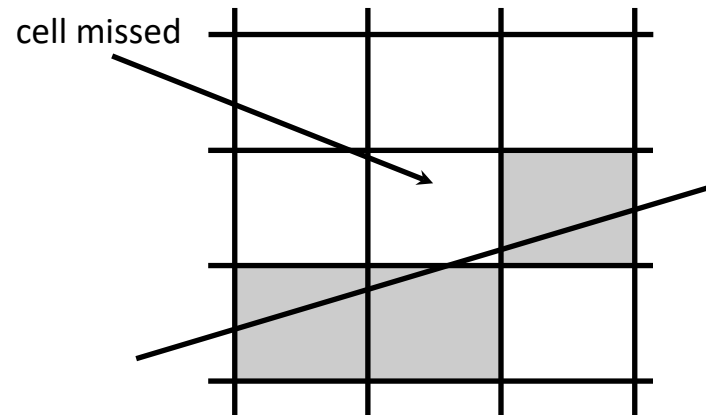


Spatial Hierarchies – Uniform Grid

- Uniform Grid
 - simple and fast traversal algorithm (e.g. line rasterization in 3D, see later on)
 - Not appropriate to handle geometry which is not equally distributed in space, high cost stepping empty cells, cannot skip empty space
 - Inadequate for handling geometries of very different sizes, no optimal cell size
- Intersection tests: two strategies
 - Extended 3D-Bresenham line drawing (rasterization)
 - follow ray from cell boundary to cell boundary

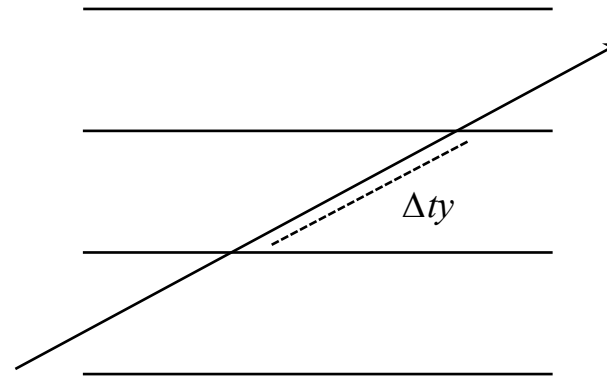
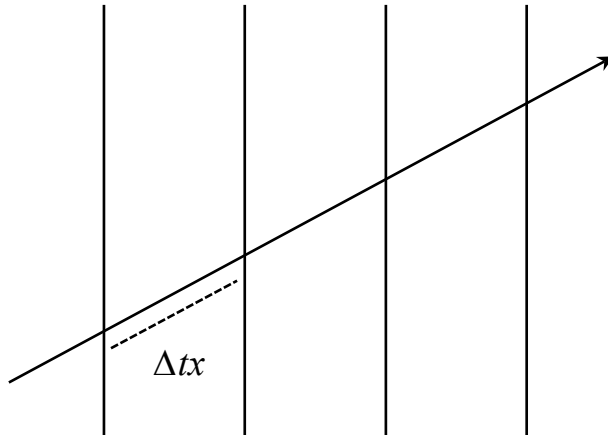
Spatial Hierarchies – Uniform Grid

- Ray Traversal with Extended Bresenham
 - major problem: rasterization method will miss some cells
 - Can be corrected by carefully looking at decider variable

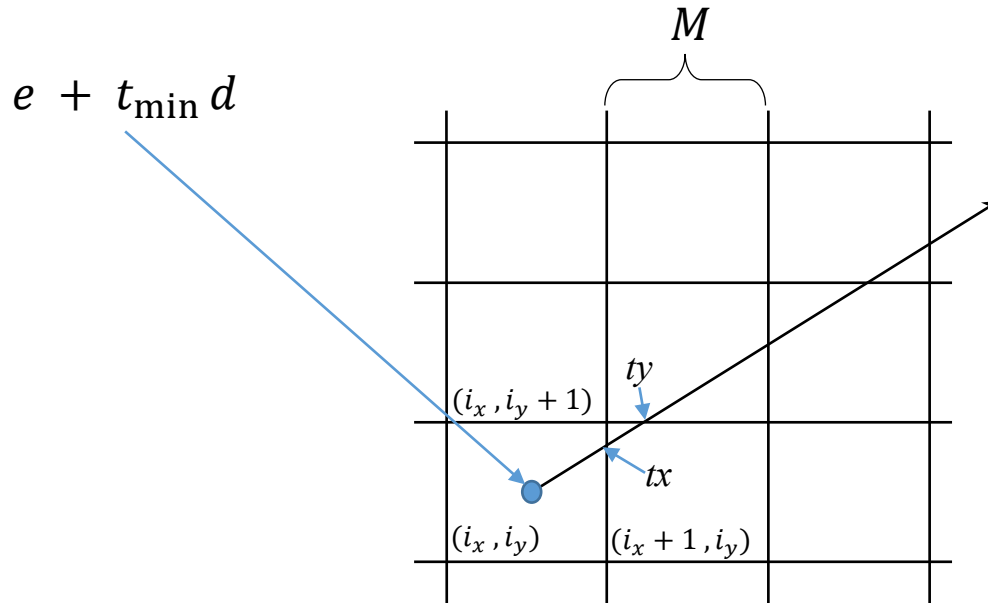


Spatial Hierarchies – Uniform Grid

- Alternative method (Amanatides 1987)
 - step from cell boundary to cell boundary
 - Key idea:
 - distance between vertical boundaries is constant, the same applies for horizontal boundaries.
 - if the ray crosses a vertical boundary, step along the x axis; if the ray crosses an horizontal boundary, step along the y axis.



Spatial Hierarchies – Uniform Grid



Spatial Hierarchies – Uniform Grid

- Method
 - maintain two variables measuring distance to next vertical and horizontal planes
 - if distance t_x to vertical plane is less than the distance t_y to horizontal plane step along x axis, else step along y axis.
 - Updates: $t_x += \Delta t_x$ and $t_y += \Delta t_y$
- Given a ray $e + t d$ with t_{\min} and t_{\max}
 - If ray has direction (d_x, d_y, d_z) then $\Delta t_x = \frac{M}{d_x}, \Delta t_y = \frac{M}{d_y}, \Delta t_z = \frac{M}{d_z}$
 - where M is the grid size
- Initialization:
 - Find grid cell of ray's starting point $e + t_{\min} d \rightarrow (i_x, i_y, i_z)$
 - Find ray parameters t_x, t_y, t_z where ray leaves corresponding slabs

Spatial Hierarchies – Uniform Grid

```
float M = grid_cell_size;

traverseGrid(float3 eye, float3 dir, float tmin, float tmax)
{
    // first grid cell (ix,iy,iz) depending on eye + tmin*dir
    int ix = ..., iy = ..., iz = ...;

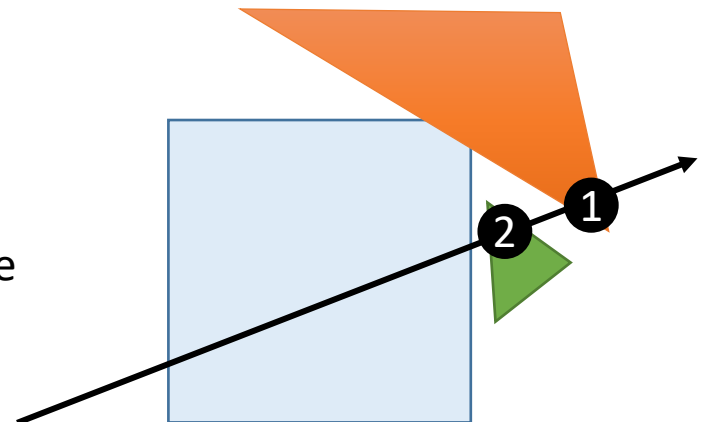
    // ray parameter along x,y,z leaving the corresponding slab
    float tx = ..., ty = ..., tz = ...;

    // step size for ray parameter along x,y,z
    float dtx = M / dir.x, dty = M / dir.y, dtz = M / dir.z;

    while (tx < tmax || ty < tmax || tz < tmax) {
        if (tx < ty && tx < tz) { // go along x
            ix++;
            intersectWithCell(eye, dir, tmin, tmax, ix, iy, iz);
            tx += dtx;
        } else if ... // other dimensions analog
    }
}
```

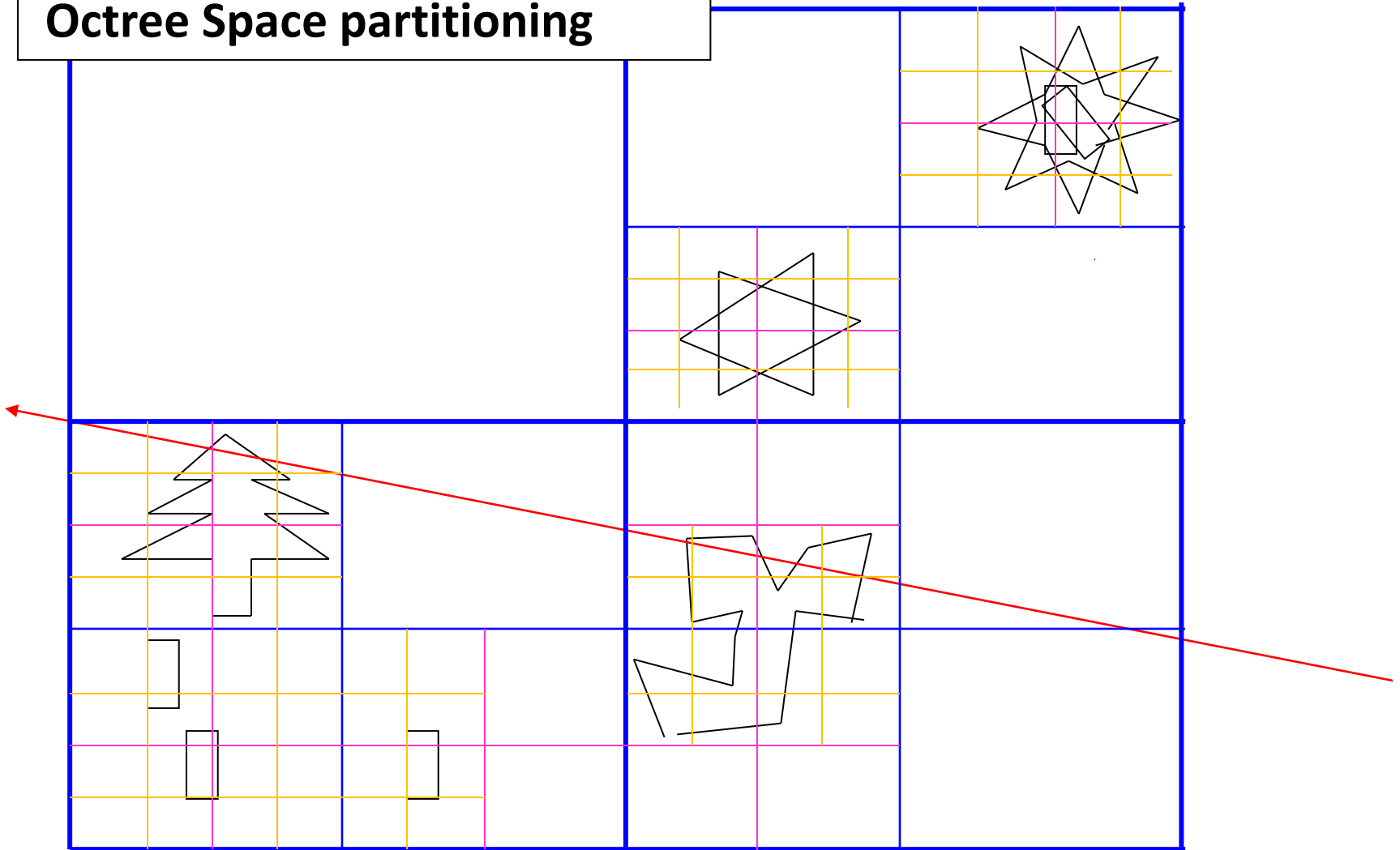
Spatial Hierarchies - Caveats

- Large triangles overlap multiple cells, so they can get checked multiple times
→ **“Mailboxing”**
 - assign a unique identifier to each ray
 - store for each triangle the index of last ray that it got checked against
 - before testing a triangle, check index to see if check was already done
 - this can fail for parallel threads, if one thread overwrites mailbox of other thread
- Be careful with first intersection!
 - see example on the right:
ray traverses blueish cell. Because the orange triangle overlaps the cell, it is checked and intersection (1) is found. However, (1) is outside the cell, so a closer intersection (2) can exist!
 - → only stop traversal when an intersection **within** the current cell has been found



Spatial Hierarchies – Octrees

Octree Space partitioning

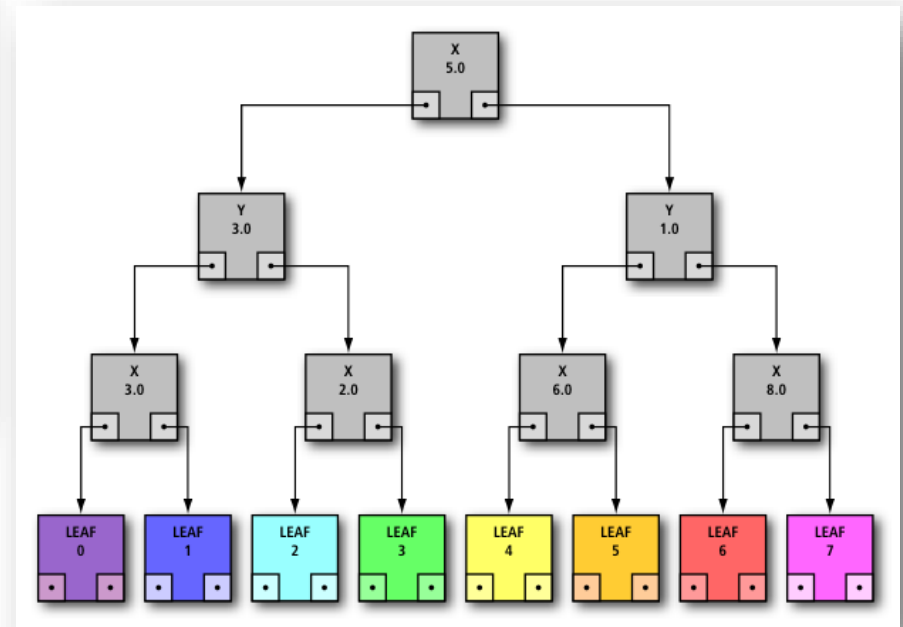
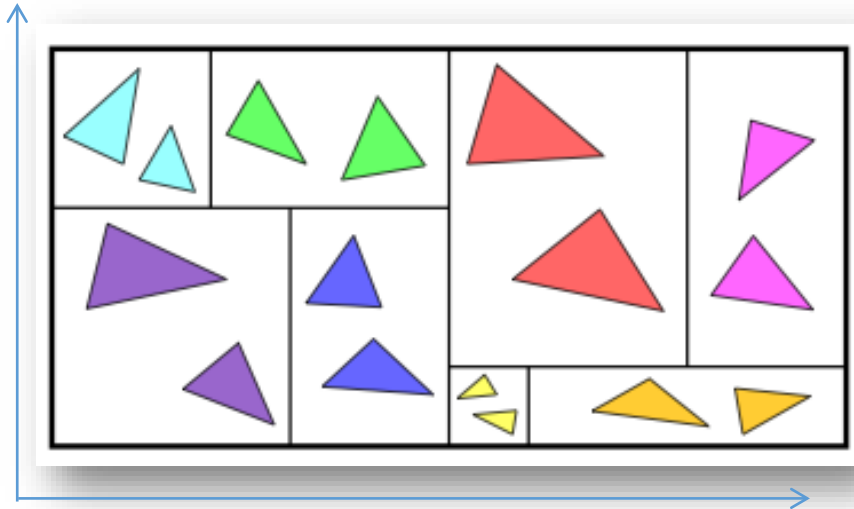


Spatial Hierarchies – Octrees

- Octree Space Partitioning
 - Cannot adapt perfectly to scene (split planes fixed)
 - traversal of children not very efficient
- Better alternative: kd-trees

Spatial Hierarchies – kd-Trees

- Example



Spatial Hierarchies – kd-Trees

- Properties

- Every node is subdivided spatially in one dimension only (x, y, and z)
- Split plane can be chosen freely → adapts well to arbitrary geometry
- Split dimension changes over levels
- very efficient recursive traversal

- Leaf-storing tree

- internal nodes only store dividing plane (splitting axis and splitting position) and reference to children nodes, but no triangles
- leaf nodes stores triangles intersecting the cells

Spatial Hierarchies – kd-Trees

- Generation: similar to BVHs
- Start with single root node with all triangles assigned
- Recursively split nodes, until a node contains less triangles than a given threshold (e.g. 10) → leaf node
- Split nodes always along the longest axis
- Decide about optimal split plane position using SAH

Spatial Hierarchies – kd-Trees

- Traversal: simple recursive approach

- as with Bounding volume hierarchies

- **But**

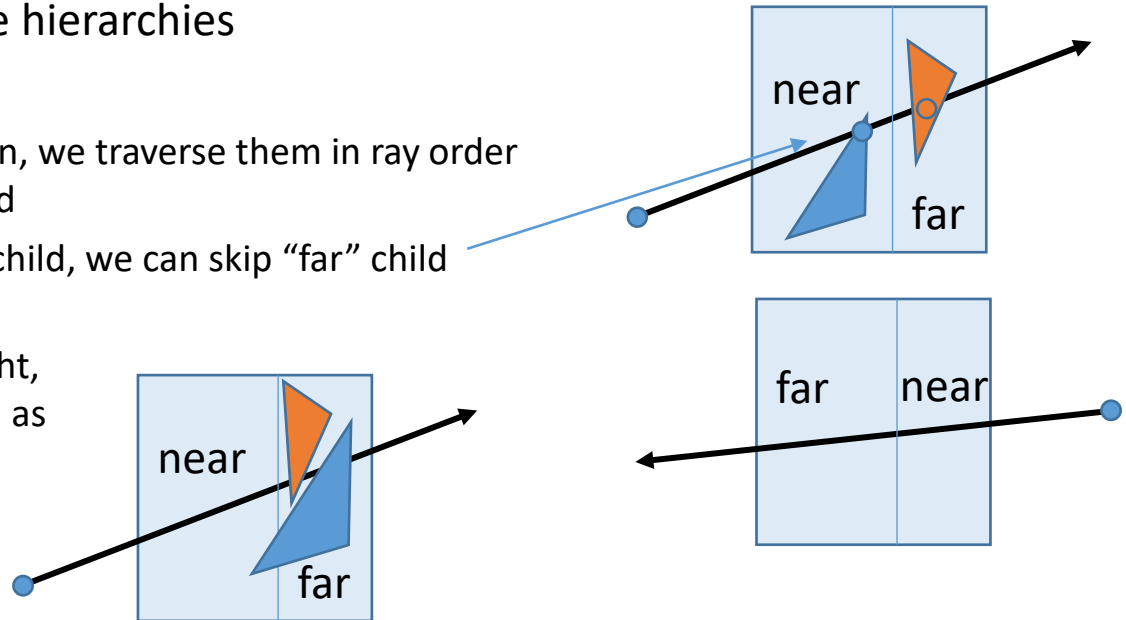
- when we traverse children, we traverse them in ray order
→ “near” child, “far” child
- If we find a hit in “near” child, we can skip “far” child
- Caveat:

In the example on the right, the blue triangle is tested as part of the near child. An intersection is found, but outside the near child.

The closer intersection with the orange triangle is thus missed

→ *only count intersection inside the node*

- **Sorting and early exit important for performance !**

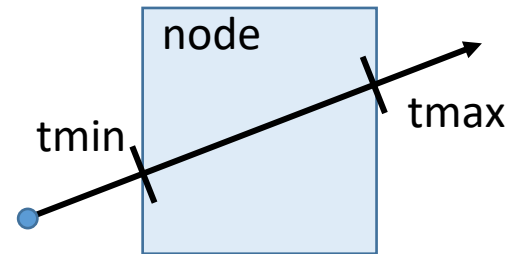


Spatial Hierarchies – kd-Trees

- Instead of recursion, it is preferable to use a stack for traversal

- An entry on the stack contains:

- a hierarchy *node* to be tested
- *tmin* of entry point, *tmax* of exit point



- We start by pushing the root node onto the stack
- We then pull nodes from the stack and process them:
 - if it is a leaf, we intersect with the triangles and stop if intersection is found
 - if it is an inner node, we intersect with the children and push the intersected children onto the stack, in near to far order
 - when stack is empty, and no intersection is found, the ray does not intersect

Spatial Hierarchies – kd-Trees

```
// test if ray intersects scene bounding box
hit = intersect(ray, scene.boundingBox);
if (hit == null)
    return NO_INTERSECTION;
stack.push(scene.boundingBox, hit.tmin, hit.tmax);

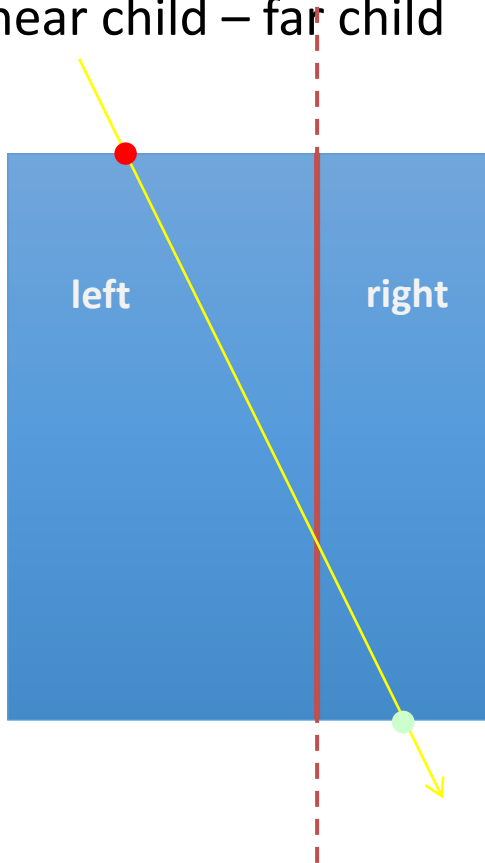
// main loop
while (!stack.empty()) {
    node, tmin, tmax = stack.pop();
    if (node.type == LEAF_NODE)
        intersect with triangles;
        stop if intersection found
    else
        intersect with children;
        push hit children on stack near to far
}
```

Spatial Hierarchies – kd-Trees

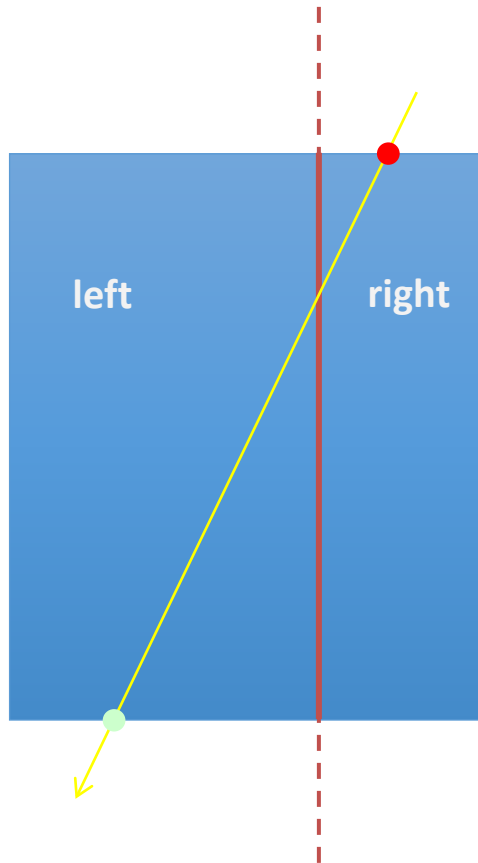
- Inner Nodes
 - determine near and far children
 - in a kd-tree we only have two children
 - order defined by sign of ray direction in split dimension
 - cases
 - ray intersect near child only: push near child onto stack
 - ray intersect far child only: push far child onto stack
 - ray intersect both children: push far child, then near child (near child will be processed first)
 - the above decision can be done very efficiently in a kd-tree!

Kd-Trees

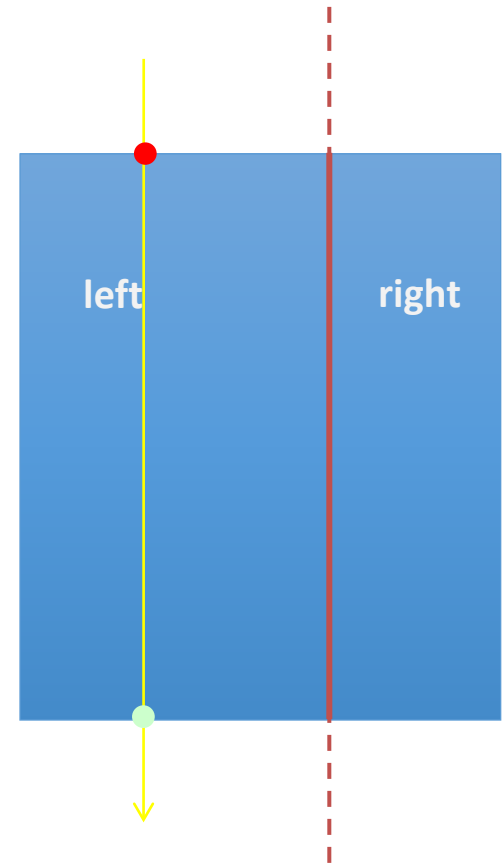
- near child – far child



$\text{ray.direction.x} > 0$
Near child: left
Far child: right



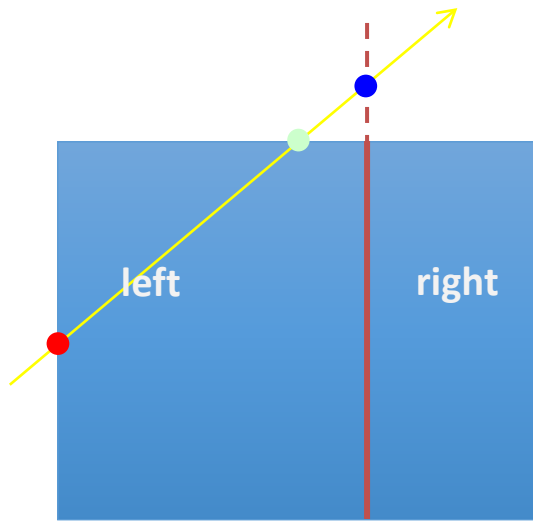
$\text{ray.direction.x} < 0$
Near child: right
Far child: left



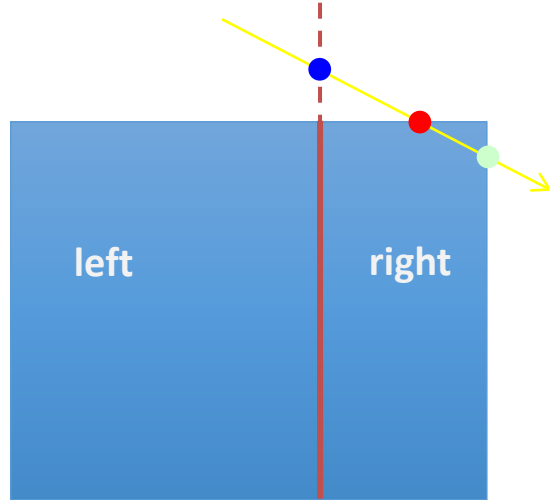
$\text{ray.direction.x} == 0$
Special case during traversal

Kd-Trees

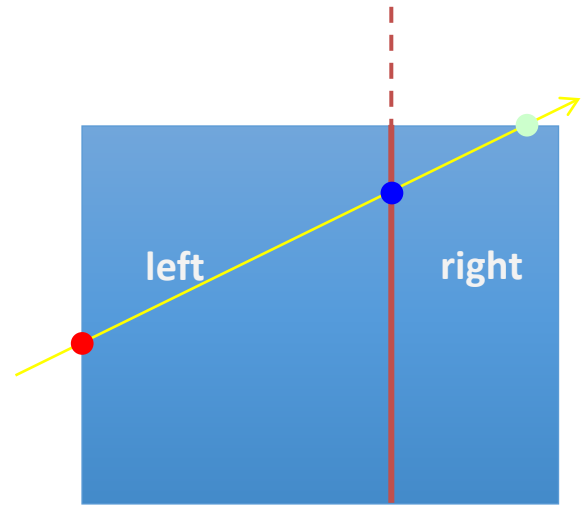
- decision which children to push: look at $tDist$ = ray parameter at split plane



Case $tDist > tMax$
Push near (left) child



Case $tDist < tMin$
Push far (right) child only



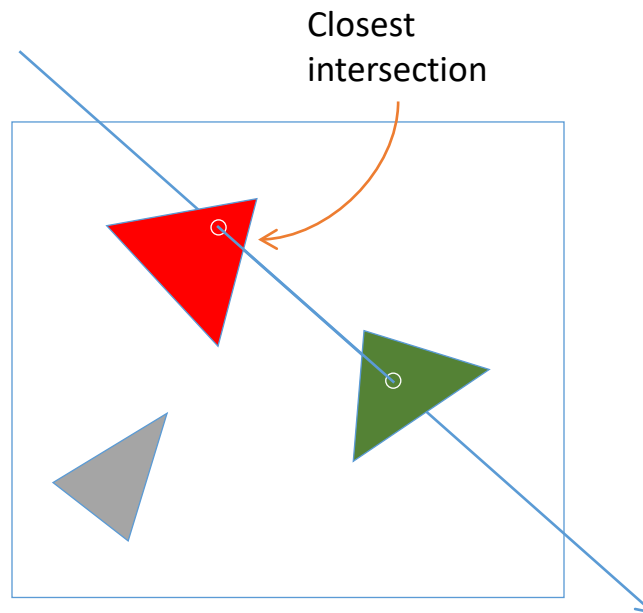
All other cases
Push both children, far (right) child first

- $tMin$, entry point
- $tMax$, exit point
- $tDist$, intersection with Kd plane

Kd-Trees

- Leaf Nodes

- intersect ray with geometric primitives in node.
- if an intersection was found inside the node (see caveat before)
 - return the closest intersection point and stop traversal
- continue traversal otherwise

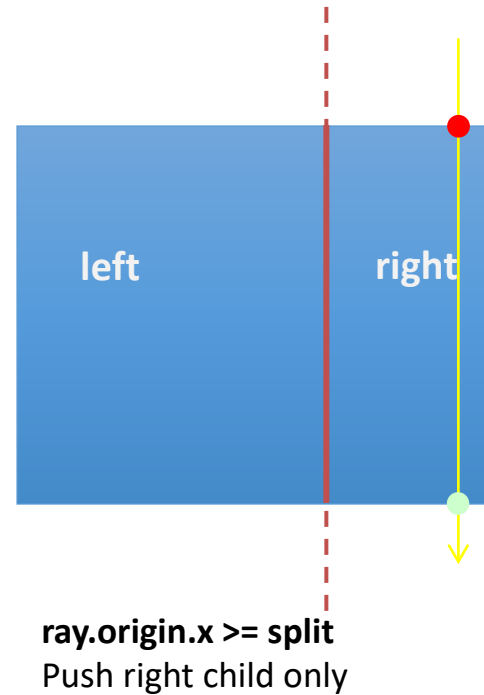
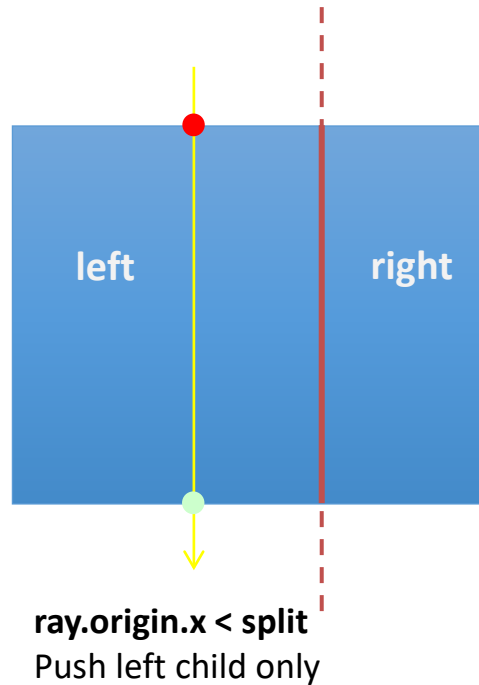


Kd-Tree

```
// main loop
while (!stack.empty()) {
    node, tmin, tmax = stack.pop();
    if (node.type == LEAF_NODE)
        intersect with triangles;
        stop if intersection found
    else
        tDist = intersection with split plane
        if (tDist > tmax)
            stack.push(near child, tmin, tmax);
        else if (tDist < tmin)
            stack.push(far child, tmin, tmax);
        else
            stack.push(far child, tmin, tdist);
            stack.push(near child, tdist, tmax);
}
```

Kd-Tree

- Inner Node: special cases
 - ray is parallel to splitting plane
 - near child depends on position of ray starting point
- Example:
split axis is x



Acceleration Structures

- Mostly used: kd-trees or BVHs
- Must be generated in a preprocess: $O(n)$, $O(n \log n)$, $O(n^2)$, ...
- but then traversal is usually $O(\log n)$
- Performance very much depends on quality of hierarchy
 - good choice of splitting plane !
 - SAH delivers good results