Lecture #18

Ray Tracing – Acceleration Structures

Computer Graphics

Winter Term 2016/17

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
    → last lecture
  - strategies and data structures to reduce the number of necessary tests
    → this lecture

Ray tracing Complexity

- Plain Ray Casting:

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

  - $O(n_p n)$ is enormous!
  - typically $n_p, n > 1.000.000$
  - 1 mio objects, 1 mio pixels → 1 trillion (10^{12}) intersection tests…
- Ray Casting: 1.000 secondary rays per pixel common
  → 10^{15} intersection tests

- But: with acceleration structures the inner loop can be accelerated to $O(\log n)$
  → entire algorithm gets $O(n_p \log n)$ → practical!

Ray tracing – Acceleration Techniques

- Fact: 9X% of the time goes into intersection tests
- Strategies for speeding up ray tracing:
  - Bounding Volumes
  - Space Partitioning
  - Ray coherence (trace bundles of rays)

- Requires one preprocessing step (for all rays)

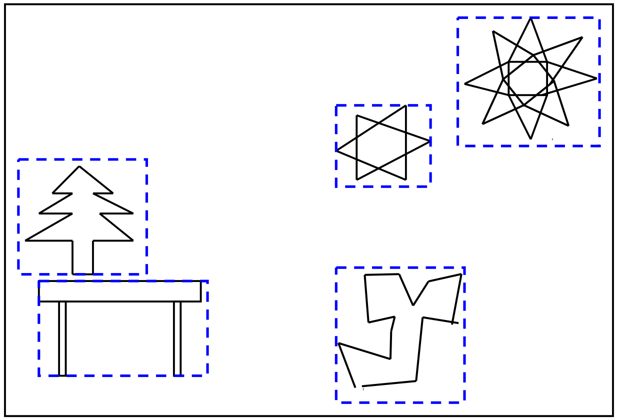
Acceleration Techniques

- Bounding volumes (BV):
- Find (geometrically simple) surrounding volumes for the complex objects
  - Spheres
  - bounding boxes
- Choose BV such that the intersection test is simple and efficient.

Acceleration Techniques

- Bounding volumes (BV) – Intersection test

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



Ray – Box Intersection

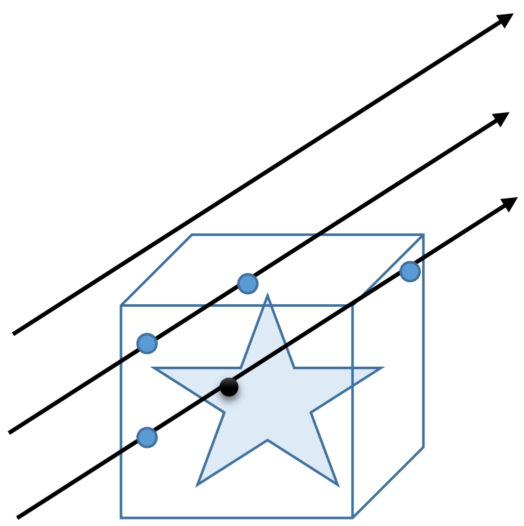
- Another important test is the intersection of a ray with a box
  → box can be a scene primitive
  → but mostly important for accelerated ray – scene interaction tests

- Idea:
  - compute a box surrounding a complex object
  - if ray misses this bounding box, no tests with complex object necessary

ray misses box → no intersection

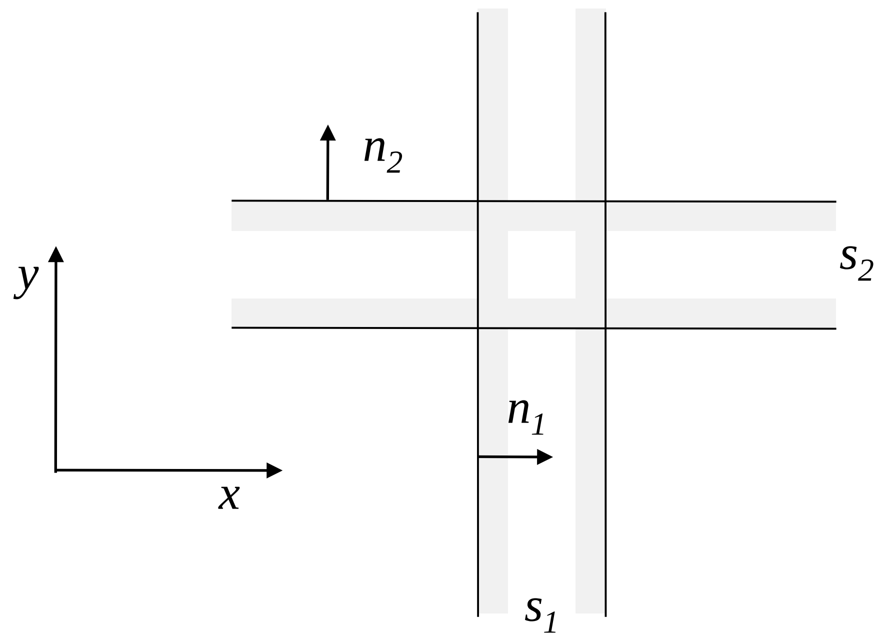
ray hits box, but no intersection with object

ray hits box, intersection with object



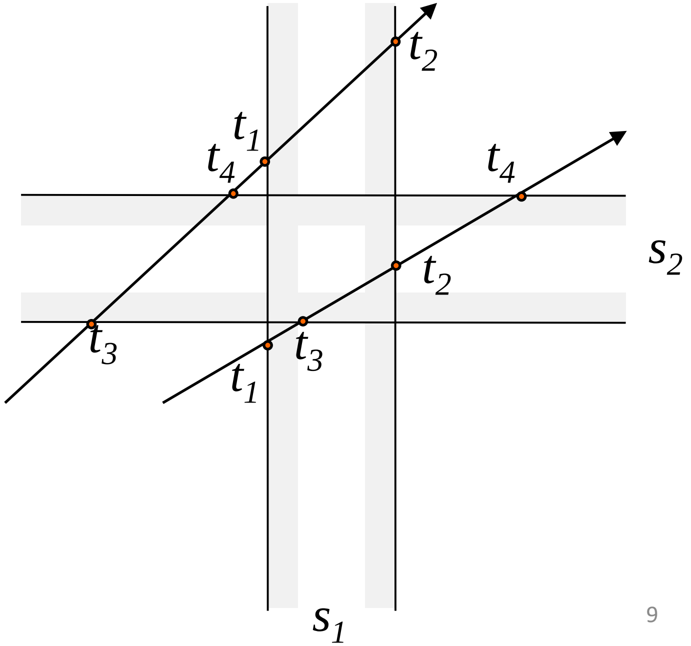
Ray – AABB Intersection

- 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

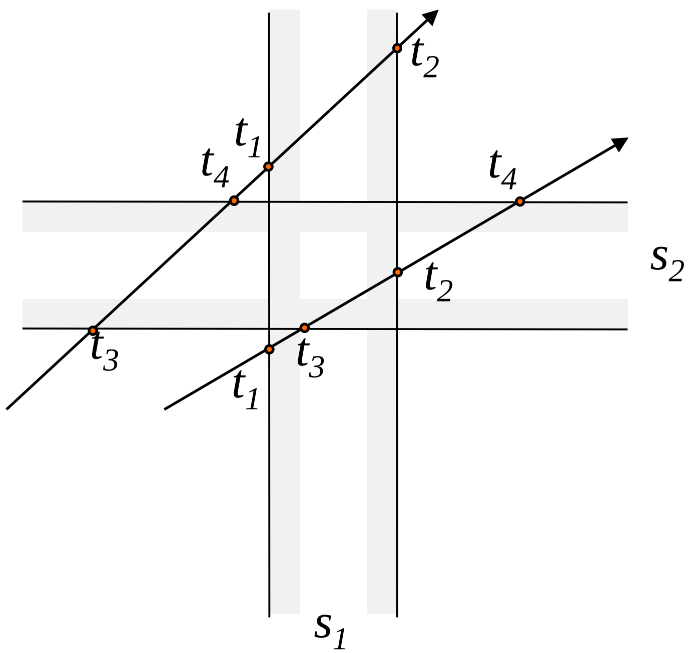


Ray – AABB Intersection

- 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 $[t_1, t_2] \cap [t_3, t_4] \neq \{\}$

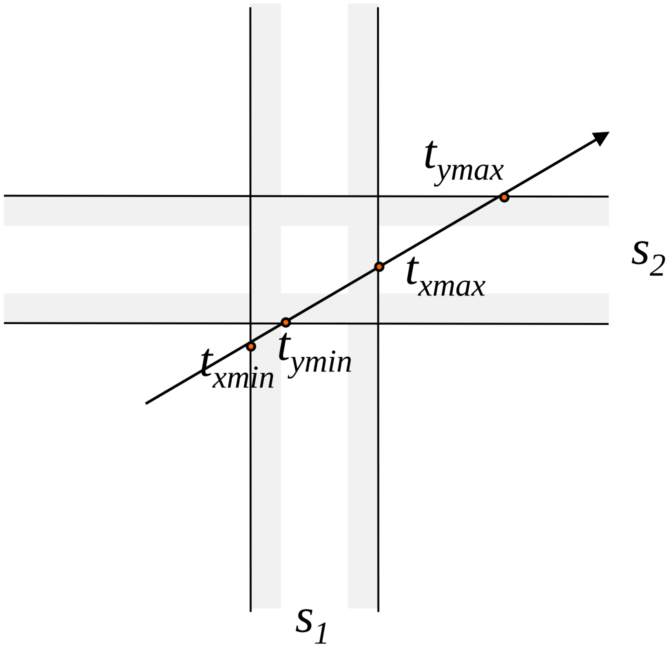


Ray – AABB Intersection

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

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

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



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

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

Ray – AABB Intersection

- 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*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;
}
```

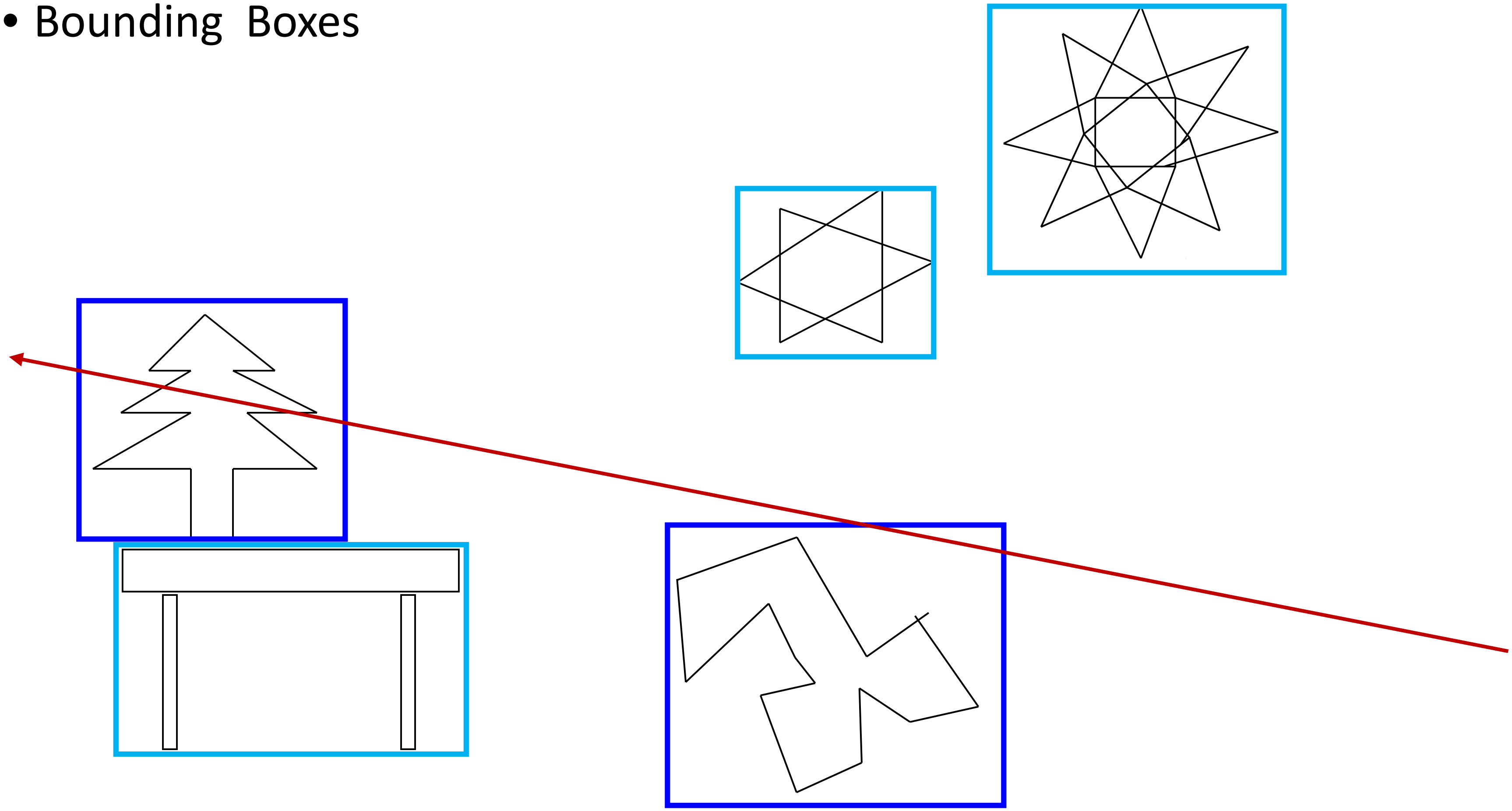
Ch. Ericson, Real-Time Collision Detection

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

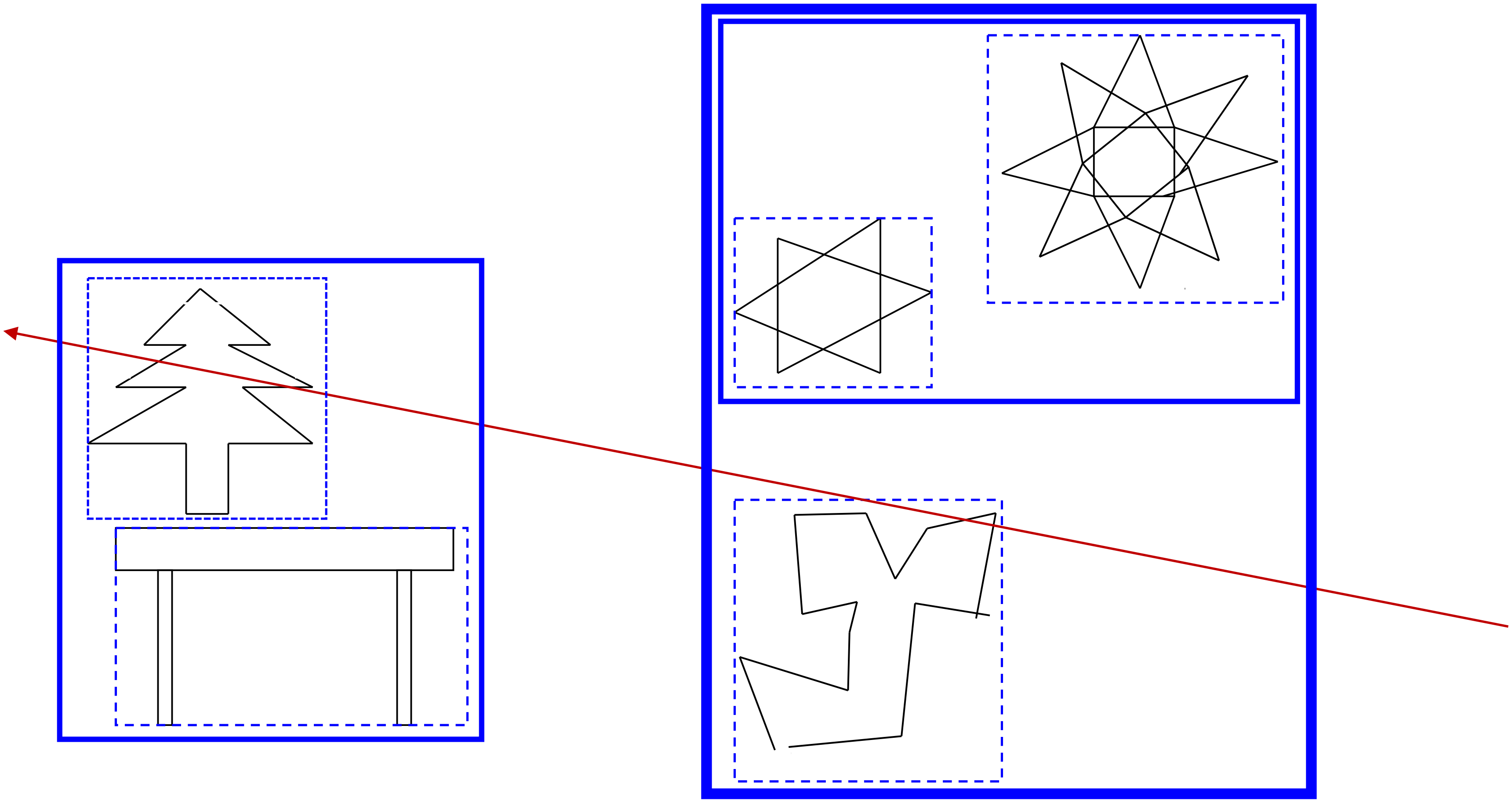
Acceleration Techniques

- Bounding Boxes



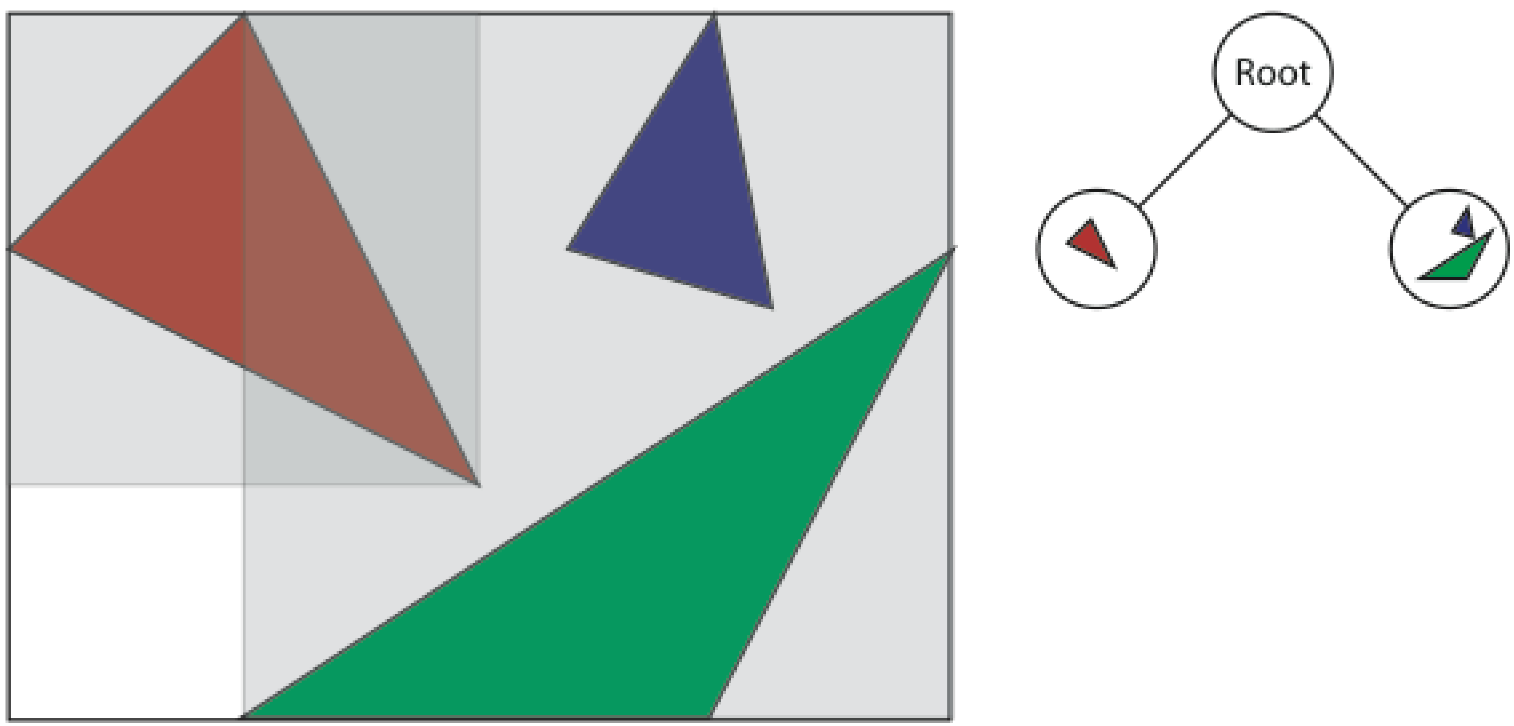
Acceleration Techniques

- Bounding Box Hierarchies



Acceleration Techniques

- BVH: Bounding Volume Hierarchy



Acceleration Techniques

- BVH (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!)
  - according to splitting strategy, objects can be split when straddling the subdivision plane

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

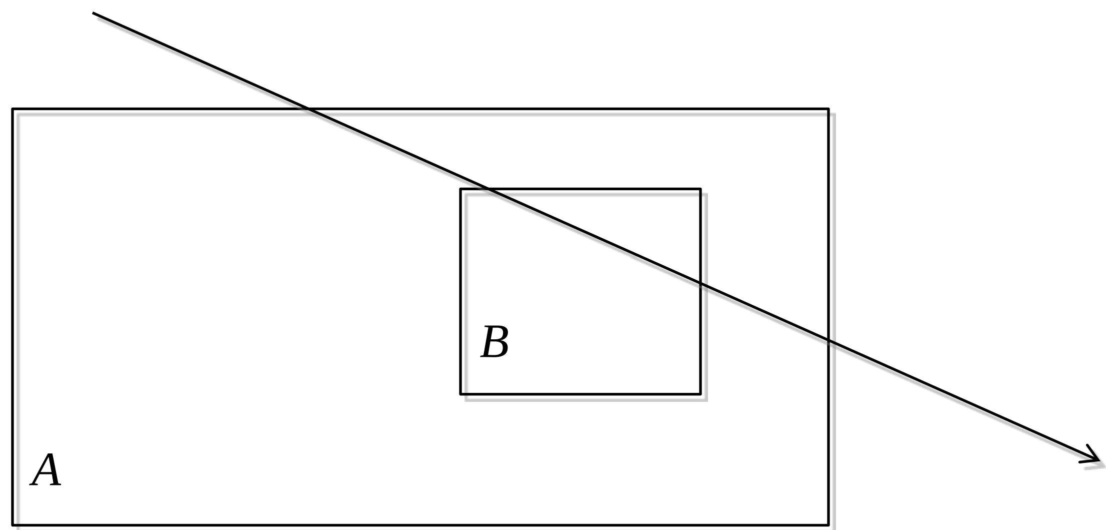
```
buildTree(Scene& scene) {
    create root node containing all triangles;
    subdivide(root);
}

subdivide(Node& node) {
    if (node.depth == MAX_DEPTH ||
        node.numTriangles <= MIN_TRIANGLES)
        return;
    compute optimal split position;
    create left and right child nodes;
    sort triangles into left and right nodes;
    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

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

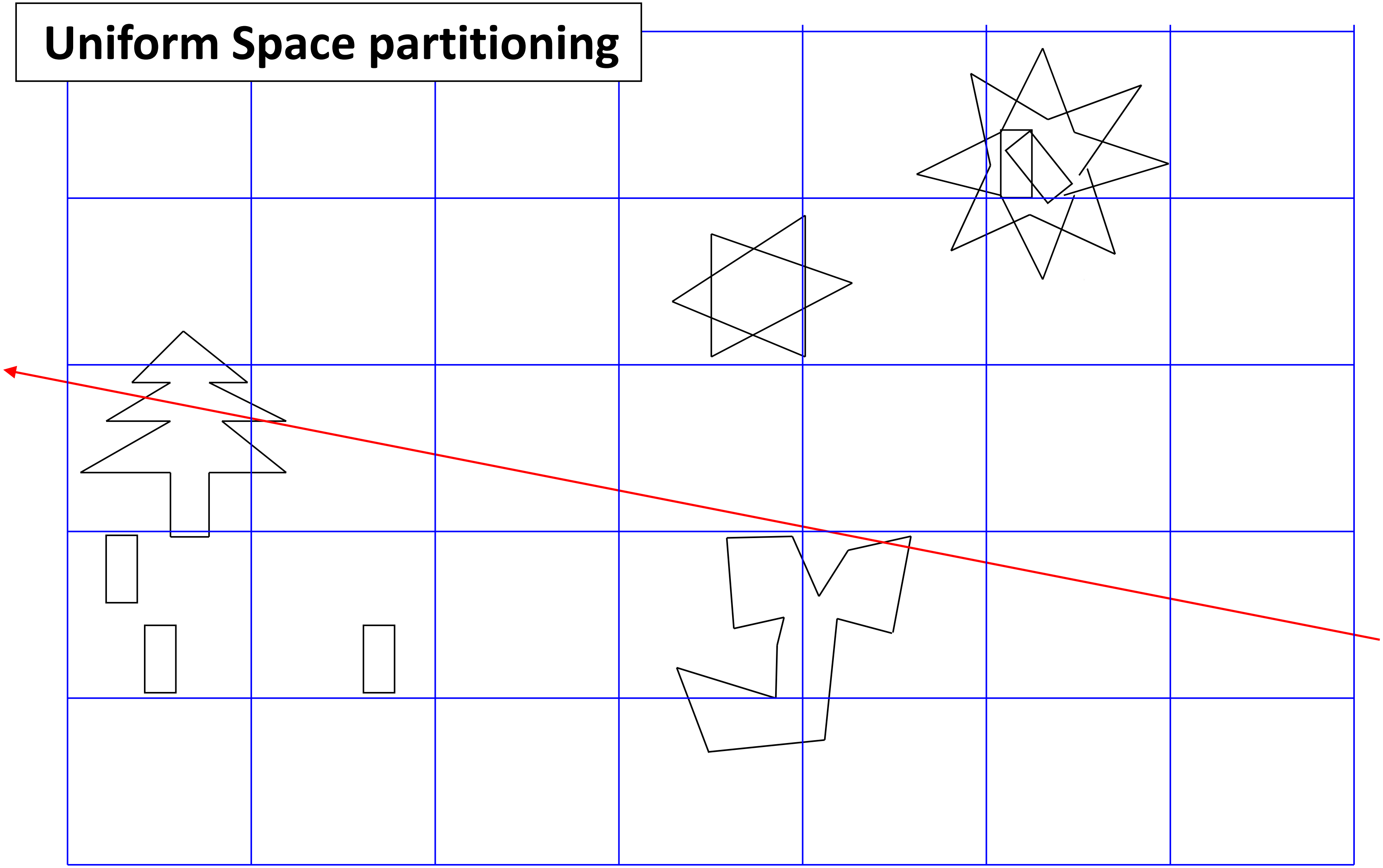
  - where P_l and P_r are the number of primitives in the left and right child nodes respectively, and B_l, B_r and B_p are the left, right and parent bounding boxes. C_T is the cost of computing a ray-primitive intersection relative to the cost of traversing a node.

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!
  - Ingo Wald: "On fast Construction of SAH-based Bounding Volume Hierarchies", 2007

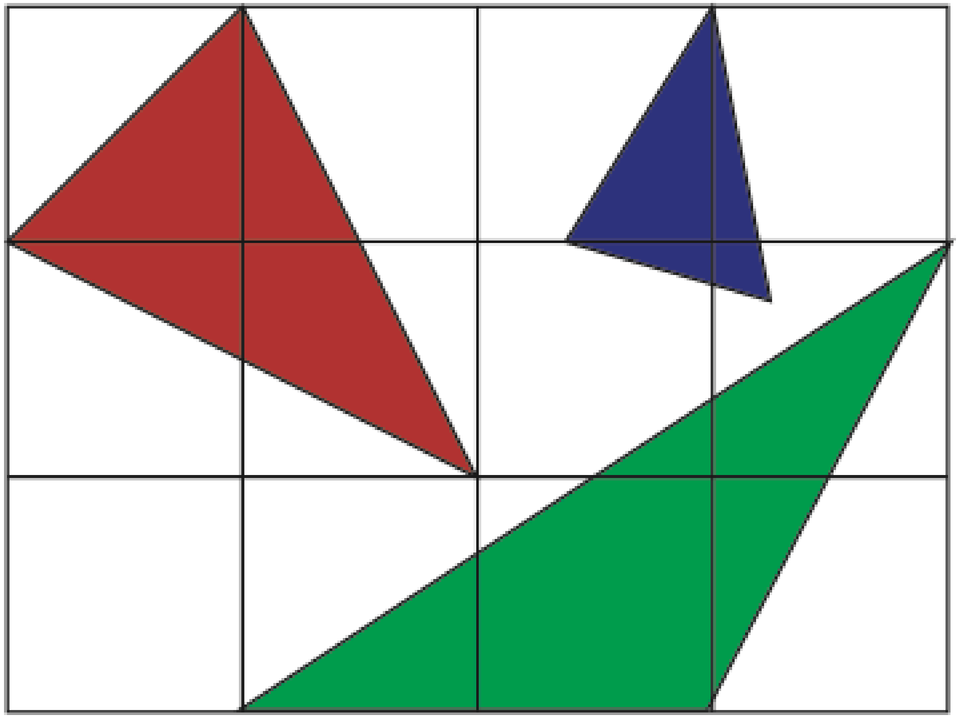
- Computing the SAH - Fast approximate SAH construction
  - generate candidate split locations for X,Y and Z
    - use minima and maxima bounding boxes of all geometric primitives
  - Evaluate SAH cost function at each location
  - select splitting plane with lowest cost
  - Compute P_l and P_r incrementally

Acceleration Techniques



Acceleration Techniques

- Uniform Grid



Acceleration Techniques

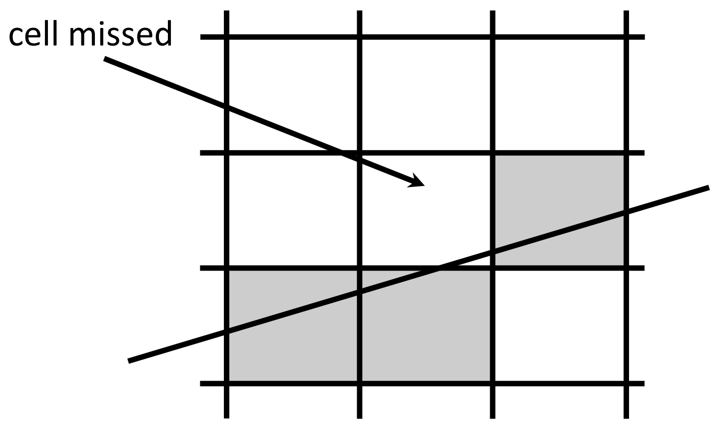
- Uniform Grid
  - simple and fast traversal algorithm (e.g. line rasterization in 3D, see later on)
  - Not appropriate to handle geometry which is not equally distributed in space, high cost stepping empty cells, cannot skip empty space
  - Inadequate for handling geometries of very different sizes,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

Ray traversal: uniform grid

- Intersection test with a uniform grid
  - two strategies
    - Extended 3D-Bresenham line drawing (rasterization)
    - follow ray from cell boundary to cell boundary

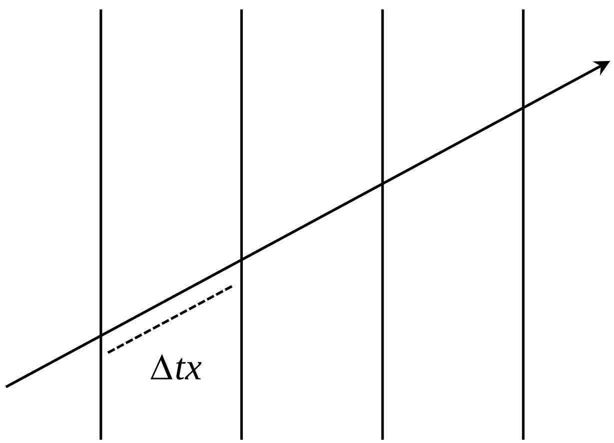
Ray traversal: uniform grid

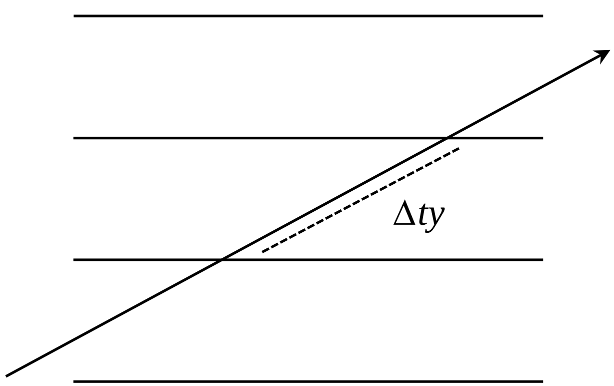
- Extended Bresenham
  - major problem: rasterization method will miss some cells
  - Can be corrected by carefully looking at decider variable



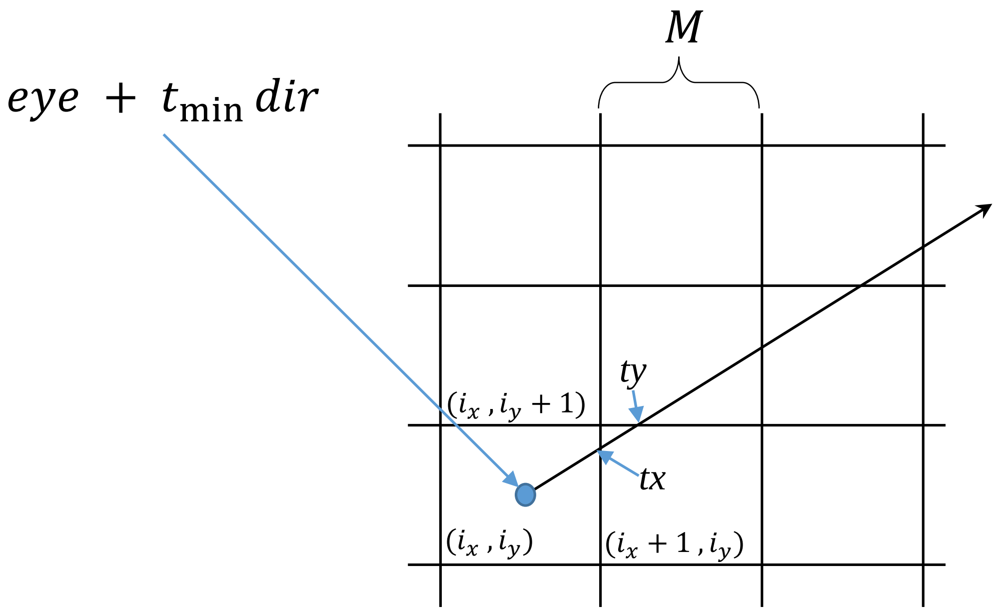
Ray traversal: uniform grid

- 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.





Ray traversal: uniform grid



Ray traversal: uniform grid

- Method
  - maintain two variables measuring distance to next vertical and horizontal planes
  - if distance tx to vertical plane is less than the distance ty to horizontal plane step along x axis, else step along y axis.
  - Updates: $t_x \ += \ \Delta t_x$ and $t_y \ += \ \Delta t_y$

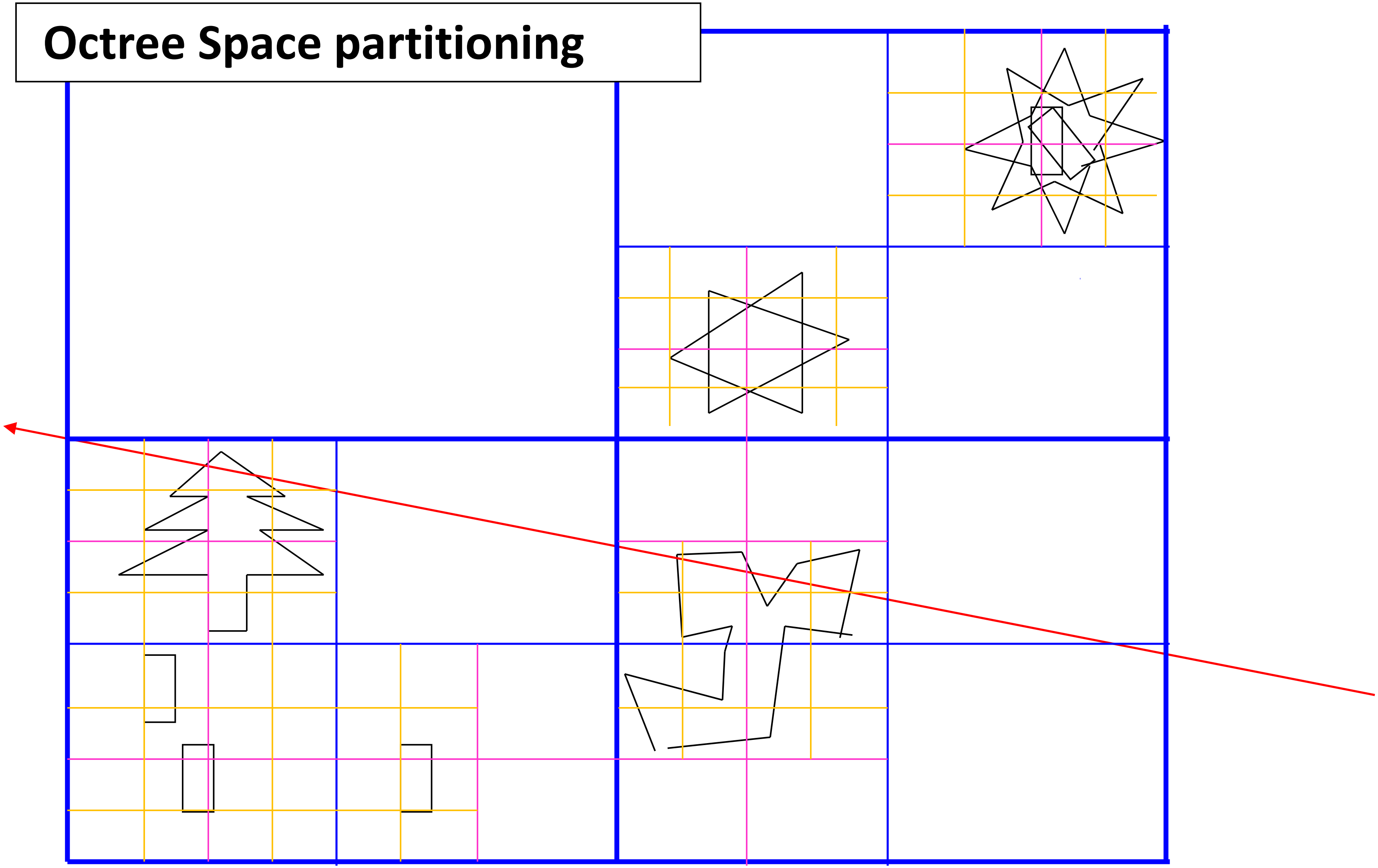
- Given a ray $eye \ + \ t \ dir$ with $t_{\min}$ and $t_{\max}$
  - If ray has direction (d_x, d_y, d_z) then $\Delta t_x = \frac{M}{d_x}, \Delta t_y = \frac{M}{d_y}, \Delta t_z = \frac{M}{d_z}$
  - where M is the grid size

- Initialization:
  - Find grid cell of ray's starting point $eye \ + \ t_{min} \ * \ dir \ \rightarrow (i_x, i_y, i_z)$
  - Find ray parameters t_x, t_y, t_z where ray leaves corresponding slabs

Ray traversal: 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 += tdx;
        else if … // other dimensions analog
    }
}
```

Acceleration Techniques

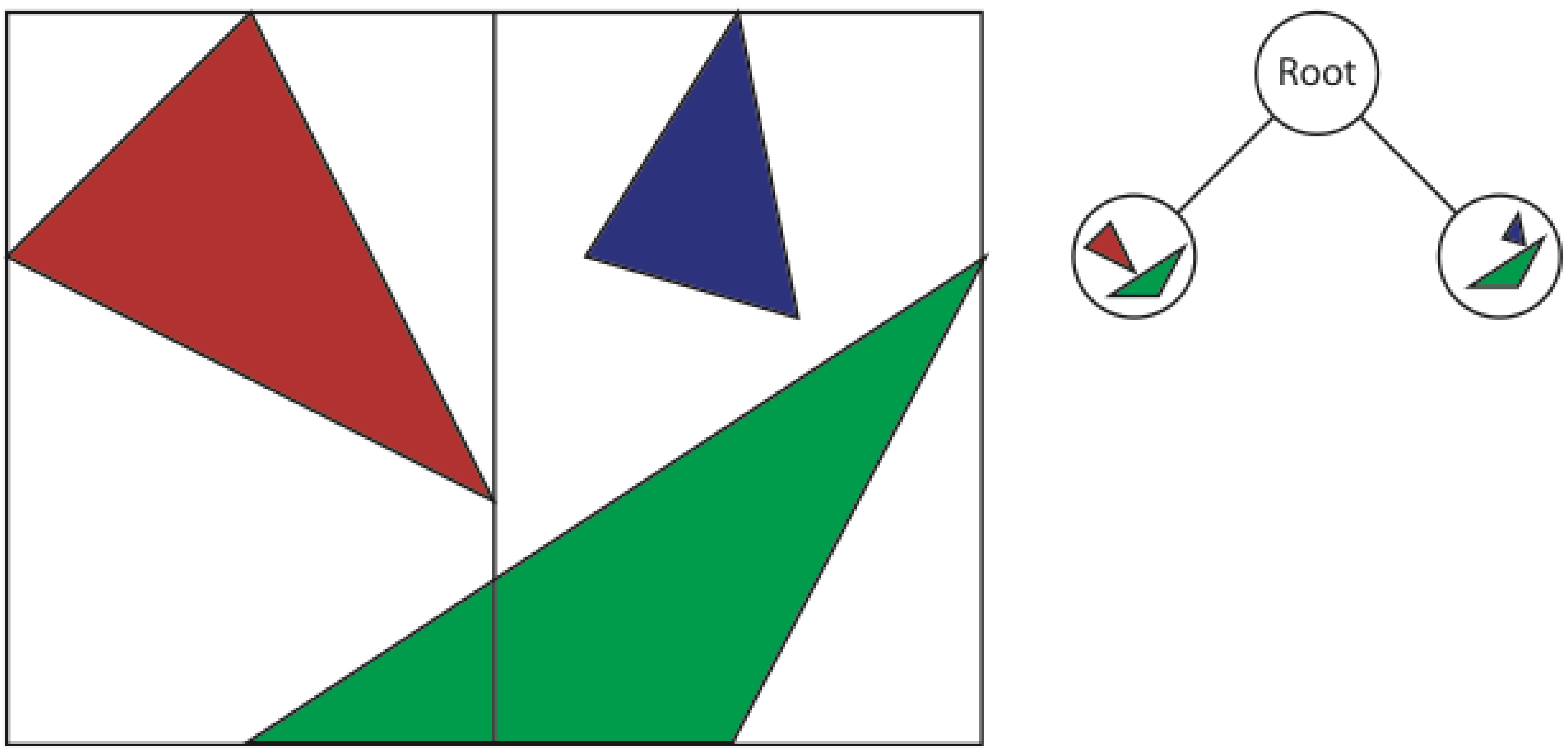


Acceleration Techniques

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

- Better alternative: kd-trees

Kd-Tree



Kd-Tree

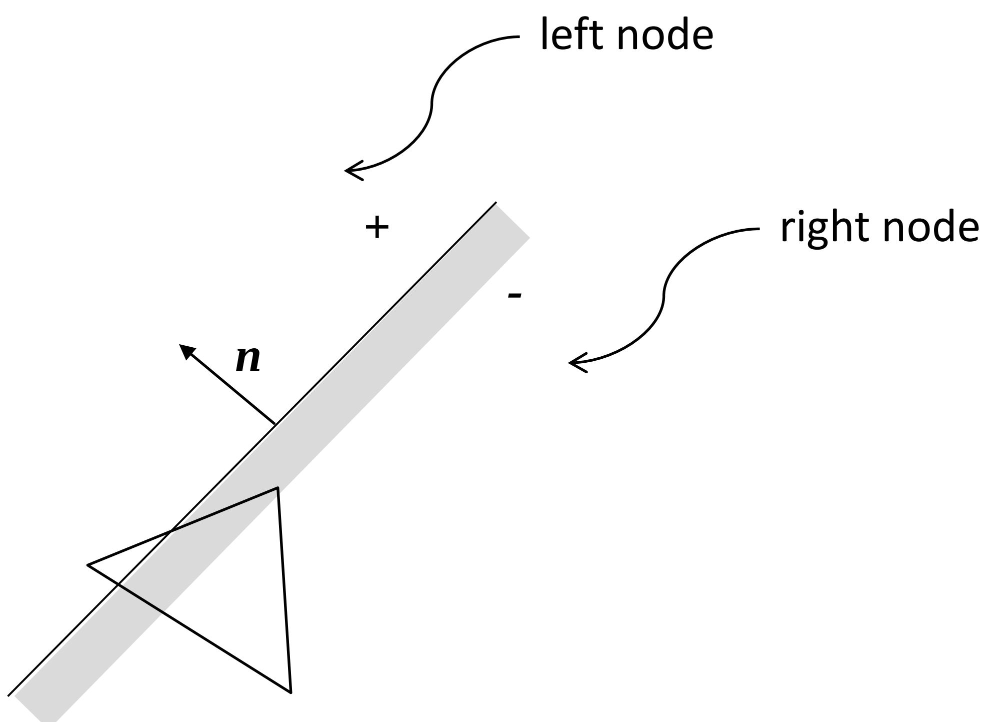
- Properties
  - Space subdivision, i.e. leaf nodes do not overlap
  - Every node is subdivided spatially in one dimension only (x, y, and z)
  - Split plane can be chosen freely → adapts well to arbitrary geometr
  - Split dimension changes over levels
  - Geometric primitives (e.g. triangles) may occur in more than one leaf node
    → contrary to BVHs !
  - nodes do not overlap
    → contrary to BVHs !
  - very efficient recursive traversal

Kd-Tree

- Leaf-storing tree
  - internal nodes store dividing plane (splitting axis and splitting position) and reference to children nodes
  - leaf nodes stores triangles intersecting the half spaces

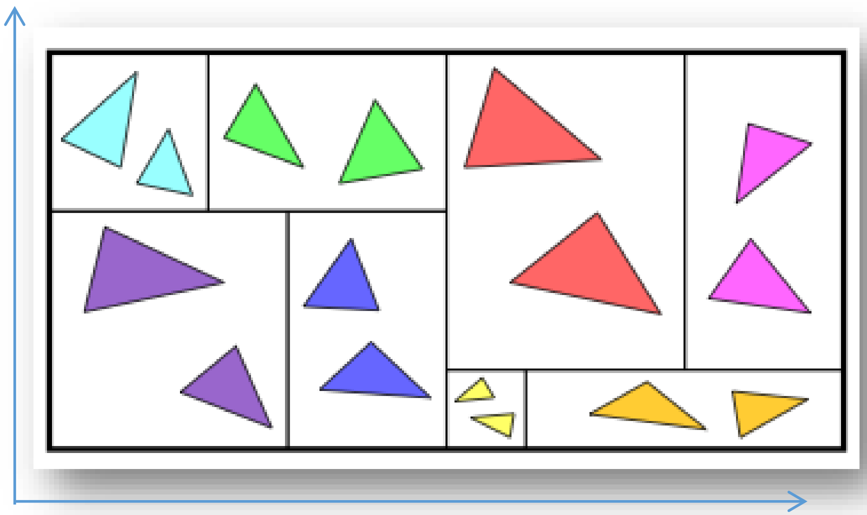
Kd-Tree

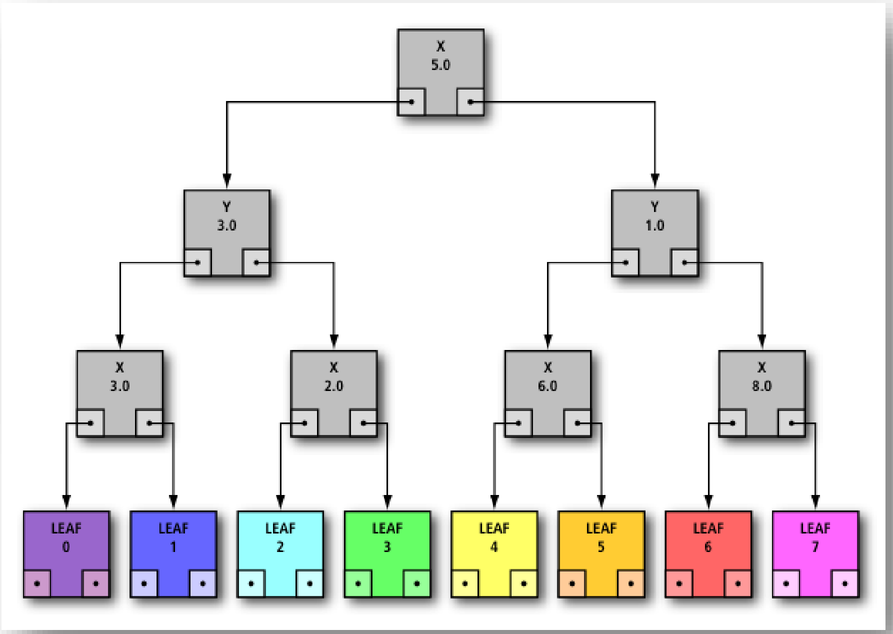
- Example
  - assign triangle to both left and right children nodes



Kd-Tree

- Example



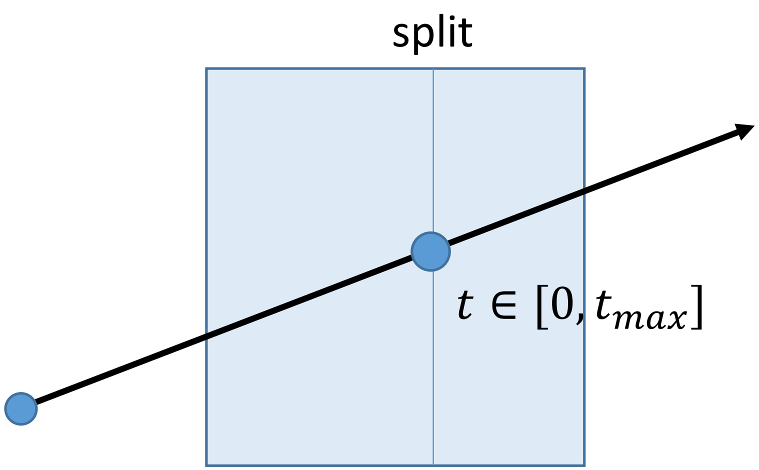


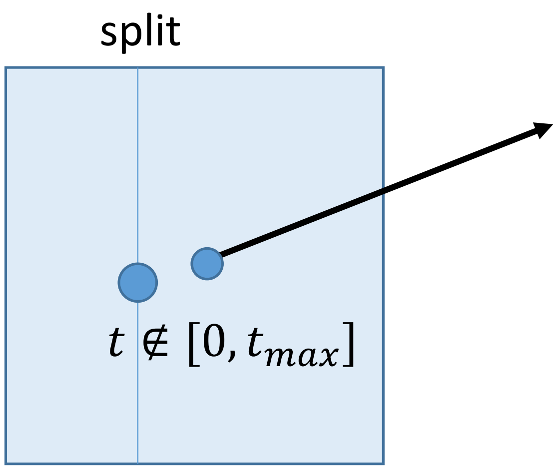
Kd-Tree

- Traversal: simple approach
  - trace ray by traversing the kd-tree and intersecting ray with splitting planes
  - line segment (ray) $p(t) = e + t\,d;\ 0 \leq t < tmax$

Kd-Tree

- Method
  - intersect line segment against node's splitting plane → t
  - test if solution satisfies $t \in [0, tmax)$
    - if t ∈ [0,tmax) descend both children recursively
      - descend children in ray order (depending on sign of direction)
        → find closest hit points first
        → allows for early exit
    - if t ∉ [0,tmax) descend child containing t = 0





Kd-Tree

- Traversal
  - The ray is traced by traversing the kd-Tree. Three stacks are used for this purpose
    - nodeStack for kd-tree nodes
    - tMinStack for parameter values at the entry points
    - tMaxStack for parameter values at the exit points

Kd-Tree

- Traversal, cont’d
  - Stacks are initialized at the root node, intersecting the bounding box of the complete scene
  - Following two cases has to be considered
    - internal nodes
    - leaf nodes
  - Stop traversal if intersection was found or stacks are empty

Kd-Tree

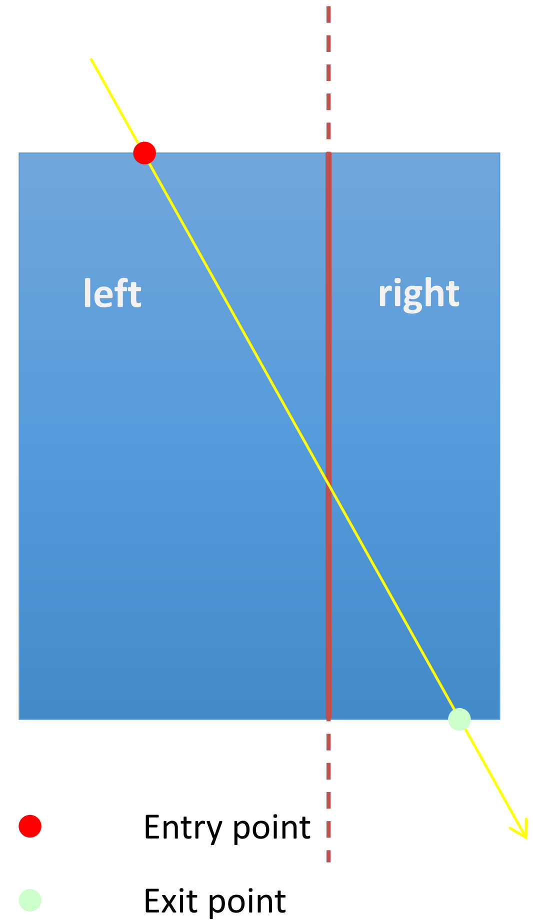
```
// test if ray intersects scene bounding box
bBoxIsect = intersect(ray, scene.boundingBox);
if (!bBoxIsect.hit)
   return NO_INTERSECTION;
nodeStack.push(kdTree.rootNode);
tMinStack.push(bBoxIsect.tMin);
tMaxStack.push(bBoxIsect.tMax);

// main loop
while (!stack.empty()) {
   node = nodeStack.top(); nodeStack.pop();
   tMin = tMinStack.top(); tMinStack.pop();
   tMax = tMaxStack.top(); tMaxStack.pop();

   // handle internal and leaf cases
   if (node.type == LEAF_NODE)
       handle leaf node;
   else
       handle internal node;
}
```

Kd-Tree

- tMin and tMax
  - tMin is the parameter value along the ray at the entry point into the current node
  - tMax is the parameter value at the exit point
  - tMin and tMax are initialized by intersecting the ray with the scene bounding box
  - If no intersection was found, the traversal returns immediately

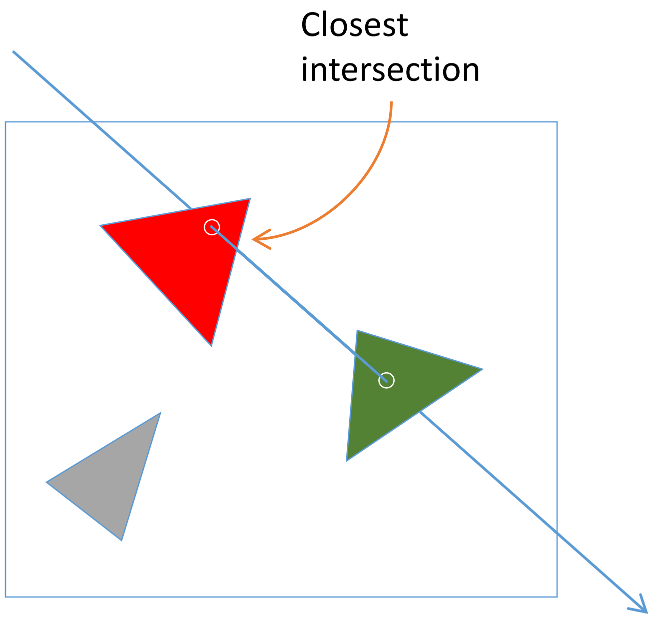
Kd-Tree

- tMin and tMax



Kd-Tree

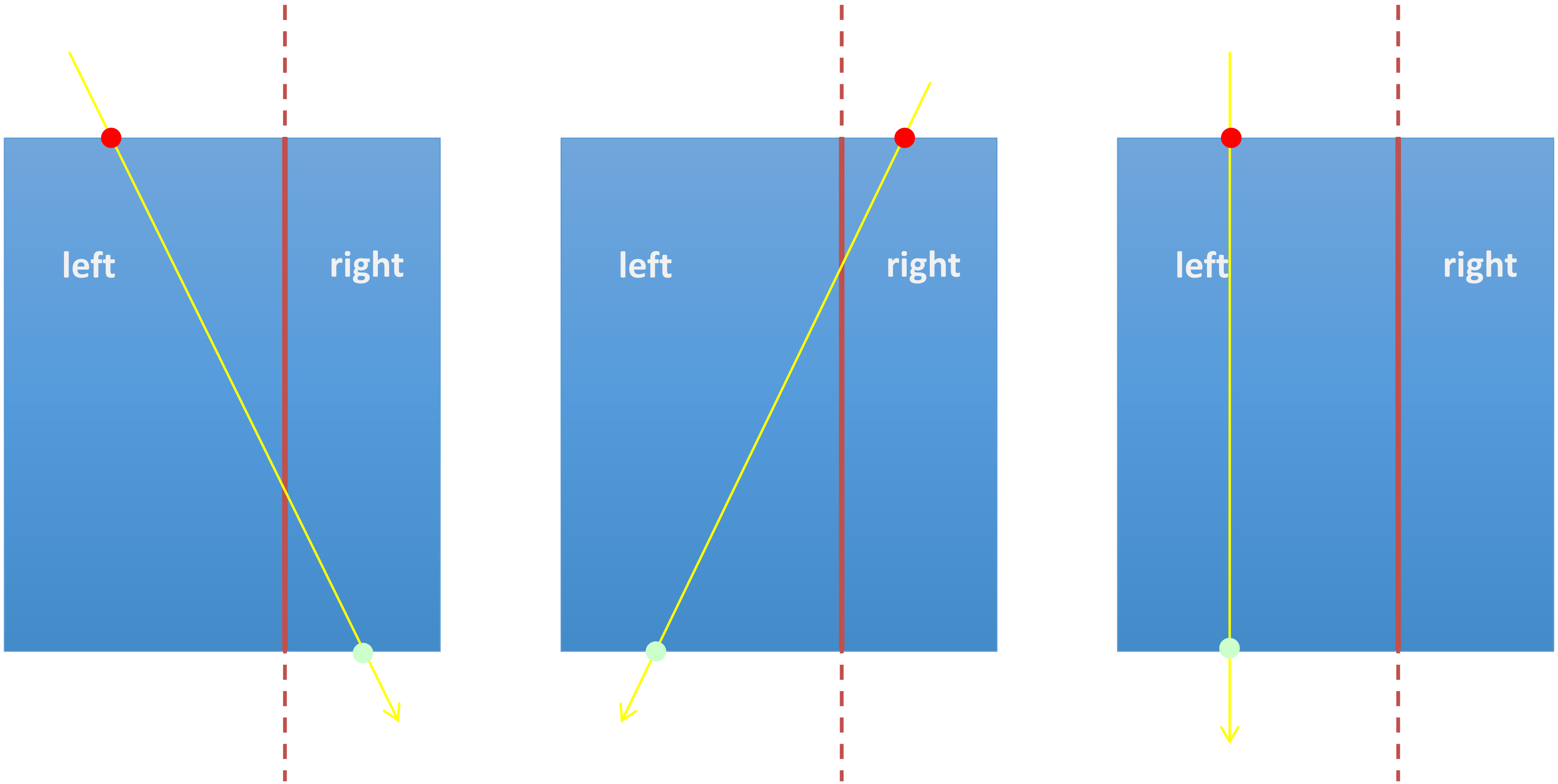
- Leaf Nodes
  - intersect ray with geometric primitives in node.
  - if an intersection was found, return the closest intersection point and stop traversal
  - continue traversal otherwise



Kd-Tree

- Internal Nodes
  - determine near and far children
  - 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)
  - continue traversal

Kd-Tree



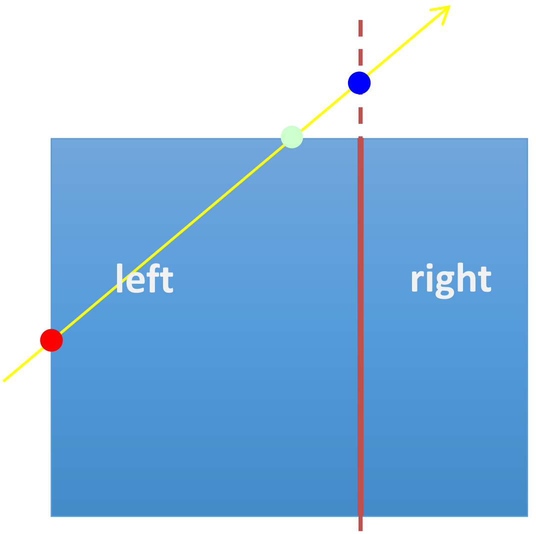
ray.direction.x > 0
Near child: left
Far child: right

ray.direction.x < 0
Near child: right
Far child: left

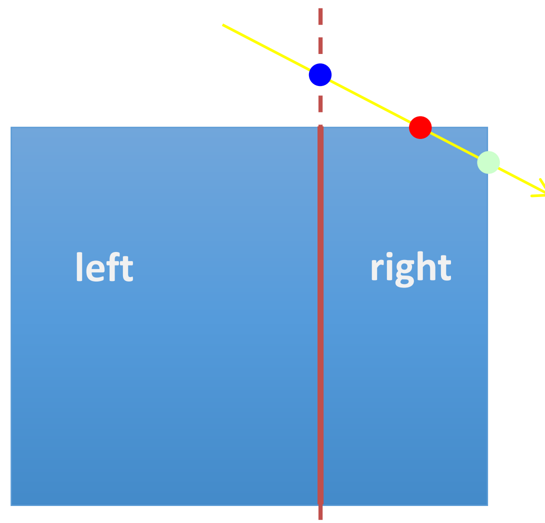
ray.direction.x == 0
Special case during traversal

Kd-Tree

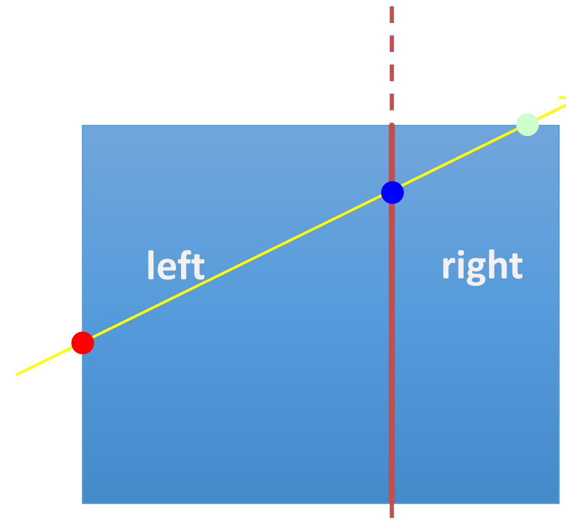
- Internal Nodes: pushing the stack



Case tDist > tMax
Push near (left) child



Case tDist < tMin
Push far (right) child only

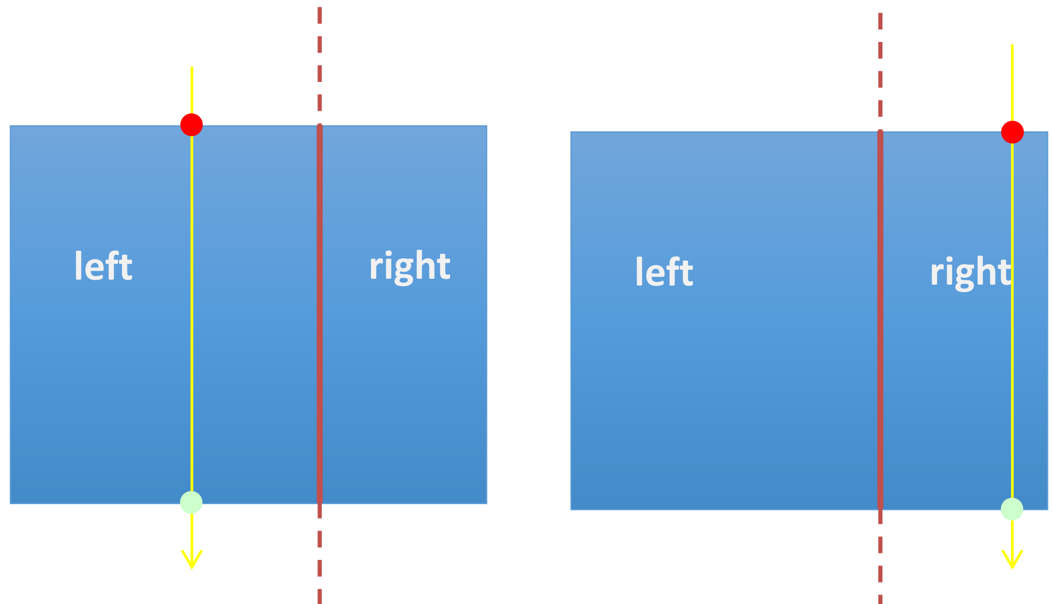


All other cases
Push both children, far (right) child first

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

Kd-Tree

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



ray.origin.x < split
Push left child only

ray.origin.x >= split
Push right child only

Acceleration Structures

- Mostly used: kd-trees or BVHs
- Must be generated in a preprocess: $O(n), O(n \log n), O(n^2), \ldots$
- 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

Acceleration Structures

- Tomorrow:
  Parallel ray traversal on CPUs and GPUs → Kai Selgrad