Computer Graphics
-Spatial Data Structures

Junjie Cao @ DLUT

Spring 2019

http://jjcao.github.io/ComputerGraphics/

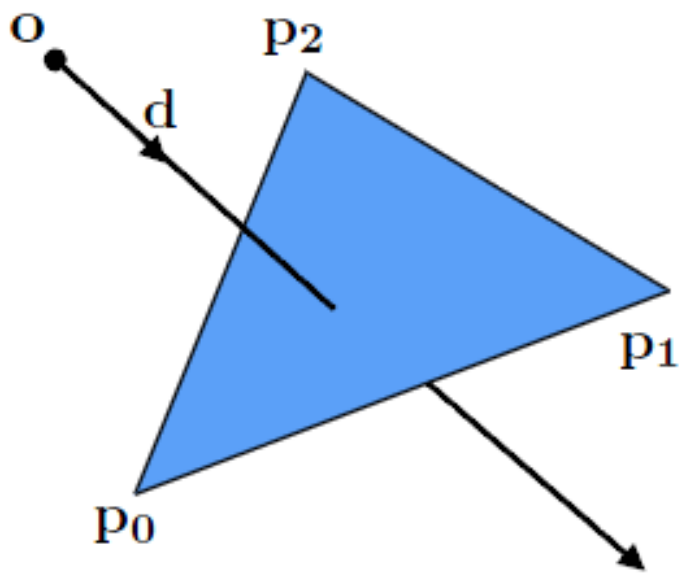
Review: ray-triangle intersection

- **Find ray-plane intersection**

Parametric equation of a ray:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

ray origin — normalized ray direction



Plug equation for ray into implicit plane equation:

$$\mathbf{N}^{\mathbf{T}}\mathbf{x} = c$$

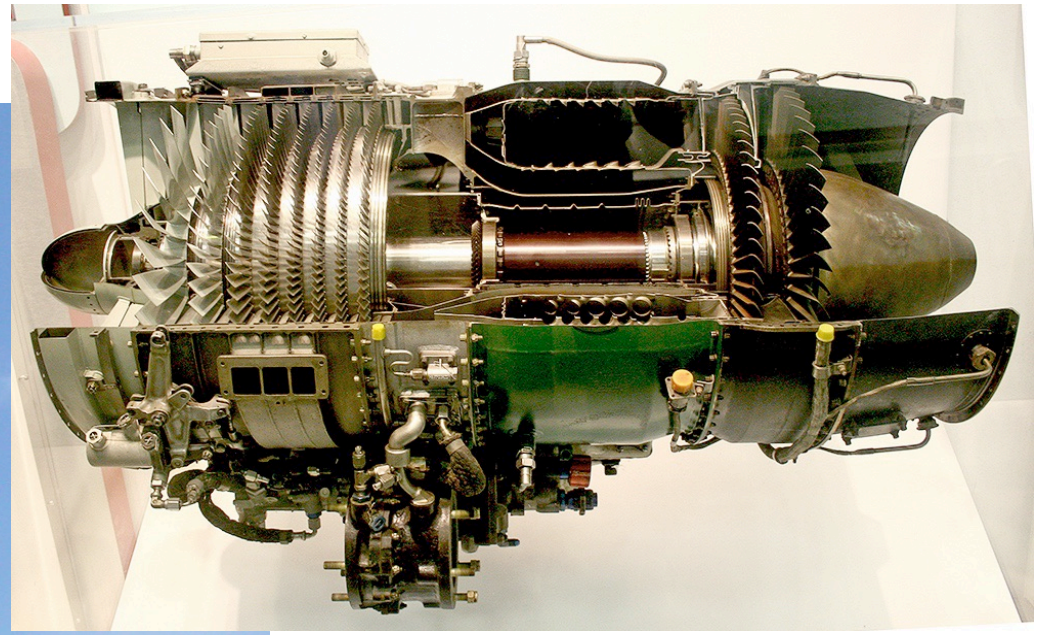
$$\mathbf{N}^{\mathbf{T}}(\mathbf{o} + t\mathbf{d}) = c$$

Solve for *t* corresponding to intersection point:

$$t = \frac{c - \mathbf{N}^{\mathbf{T}}\mathbf{o}}{\mathbf{N}^{\mathbf{T}}\mathbf{d}}$$

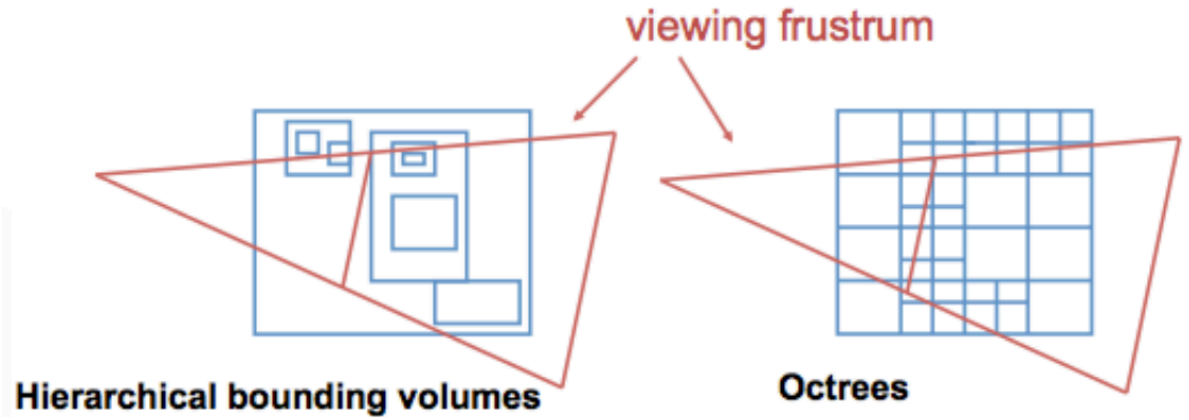
- **Determine if point of intersection is within triangle**

Complexity of geometry



Types of Queries

- Graphic applications **often require** spatial queries
  - Find the k points closer to a specific point p (k-Nearest Neighbours, knn)
  - Is object X intersection with object Y? (Intersection)
  - What is the volume of the intersection between two objects?
- **Brute force search is expensive**
  - Instead, you can solve these queries with an initial preprocessing that creates a data structure which supports efficient queries
  - **Speed-up of 10x, 100x, or more**



Clipping with spatial data structure

Accelerating Geometric Queries

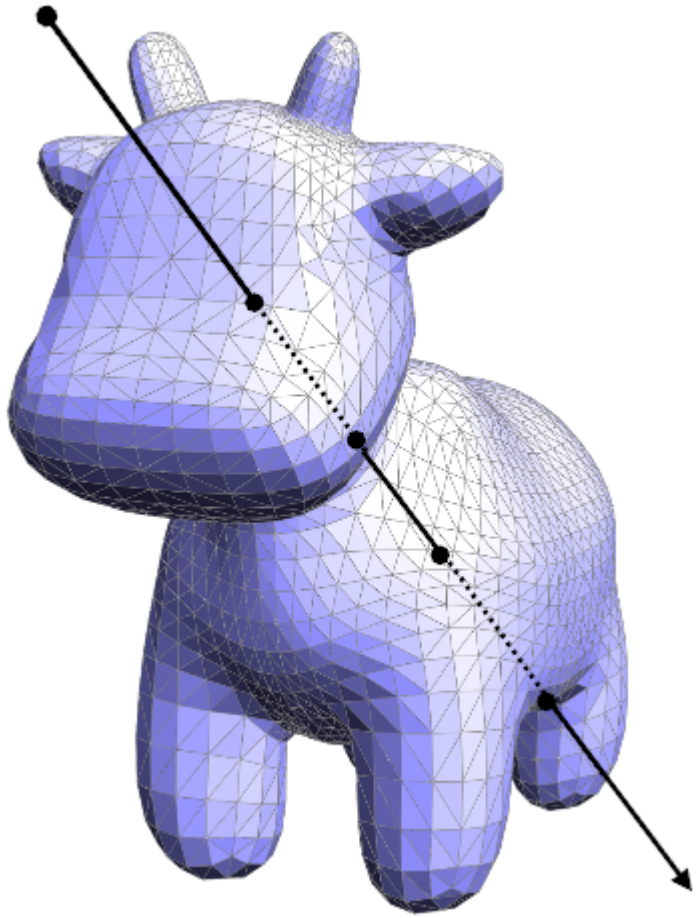
Ray-primitive queries

- **Given primitive p:**
- **p.intersect(r) returns value of *t* corresponding to the point of intersection with ray *r***
- **p.bbox() returns axis-alighed bounding box of the primitive**
  - tri.bbox():
    - tri_min = min (p0, min(p1,p2))
    - tri_max = max (p0, max(p1,p2))
    - Return bbox(tri_min, tri_max)

Ray-scene intersection

- **Find the first primitive the ray hits**
  - Given a scene defined by a set of *N* primitives and a ray *r*, find the closest point of intersection of *r* with the scene

```
p_closest = NULL
t_closest = inf
for each primitive p in scene:
    t = p.intersect(r)
    if t >= 0 && t < t_closest:
        t_closest = t
        p_closest = p
```



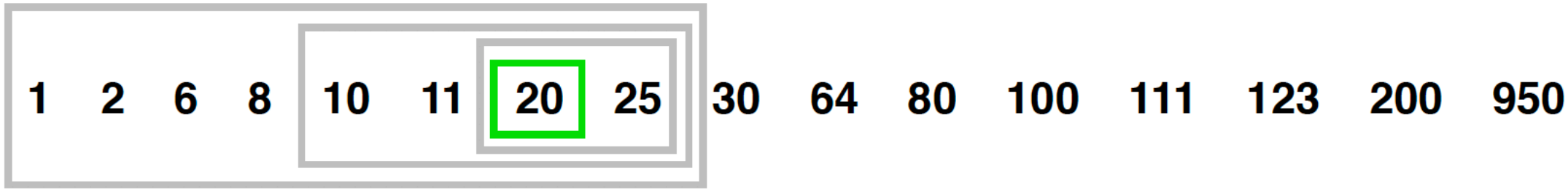
Complexity: **O(N)**

A simpler problem

- Imagine I have a set of integers S
- Given an integer *k=18*, find the element in S that is closest to *k:*

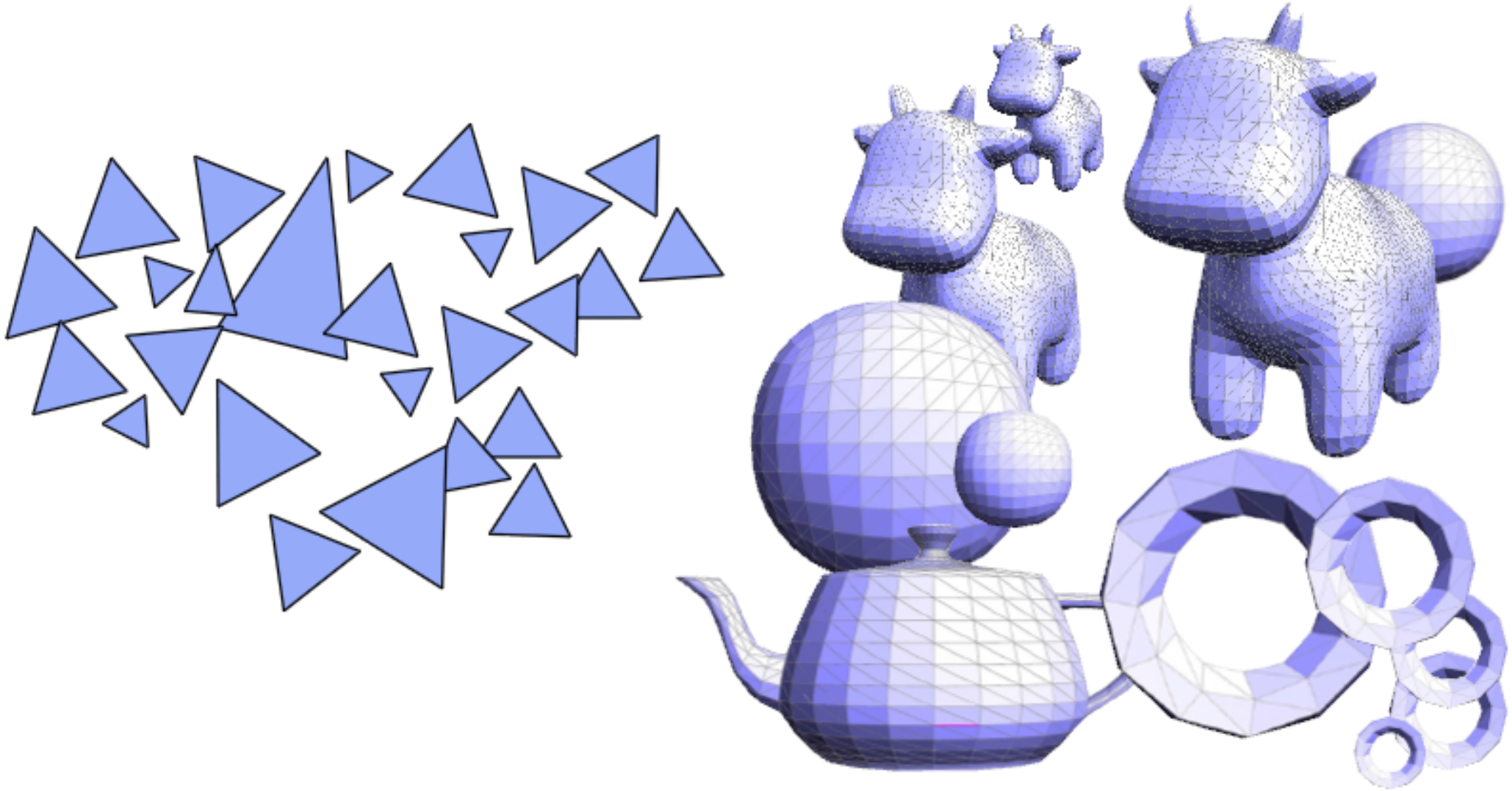
10 123 2 100 6 25 64 11 200 30 950 111 **20** 8 1 80

- What's the cost of finding k in terms of the size N of the set?
- **Can we do better?**
- Suppose we first sort the integers:



- **How much does it now cost to find k (including sorting)?**
- **Cost for just ONE query: O(n log n)**
- **Amortized cost: O(log n)**

How do we organize scene primitives to enable fast ray-scene intersection queries?



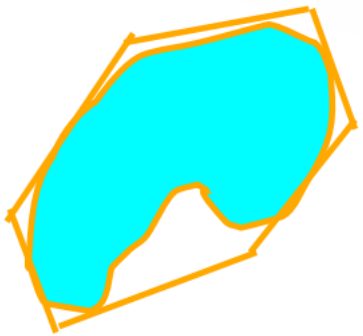
Two main ideas

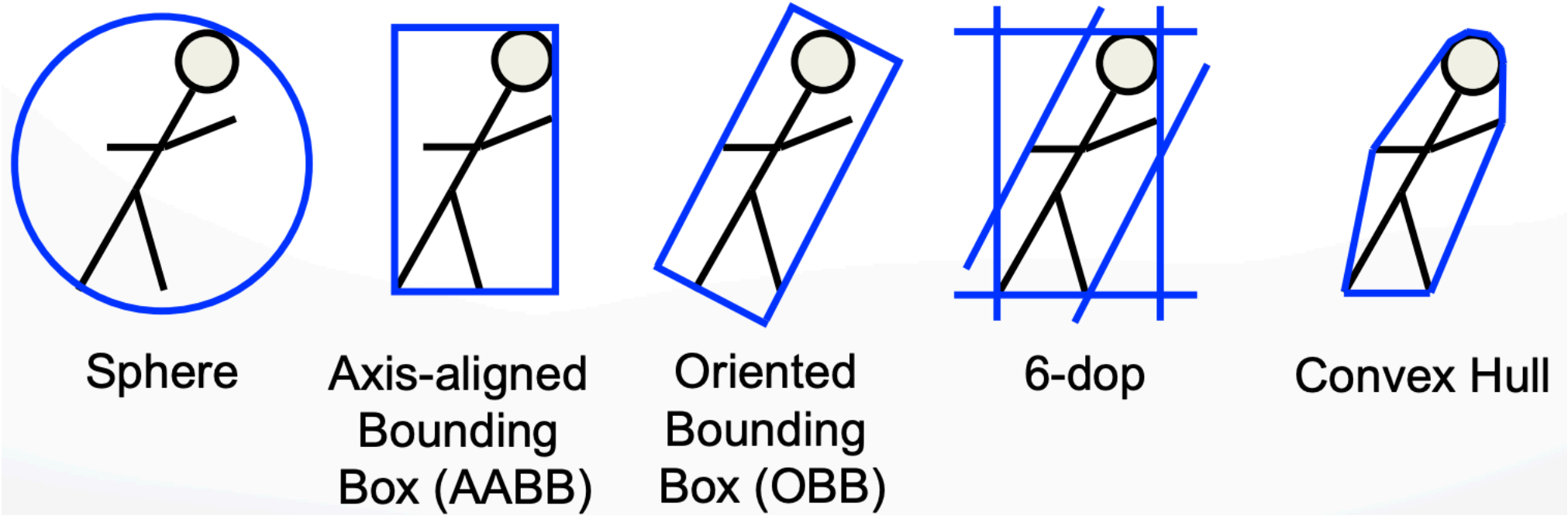
- **object partitioning** (Bounding Volume Hierarchies)
  - objects are divided into disjoint groups,
  - but the groups may end up overlapping in space.

- **space partitioning** (…)
  - space is divided into separate partitions,
  - but one object may have to intersect more than one partition.

Wrap complex objects in simple ones

- Does ray intersect bounding box?
  - No: does not intersect enclosed objects
  - Yes: calculate intersection with enclosed objects
- Common types:





Ray-axis-aligned-box intersection

- What is ray's closest/farthest intersection with axis-aligned box?

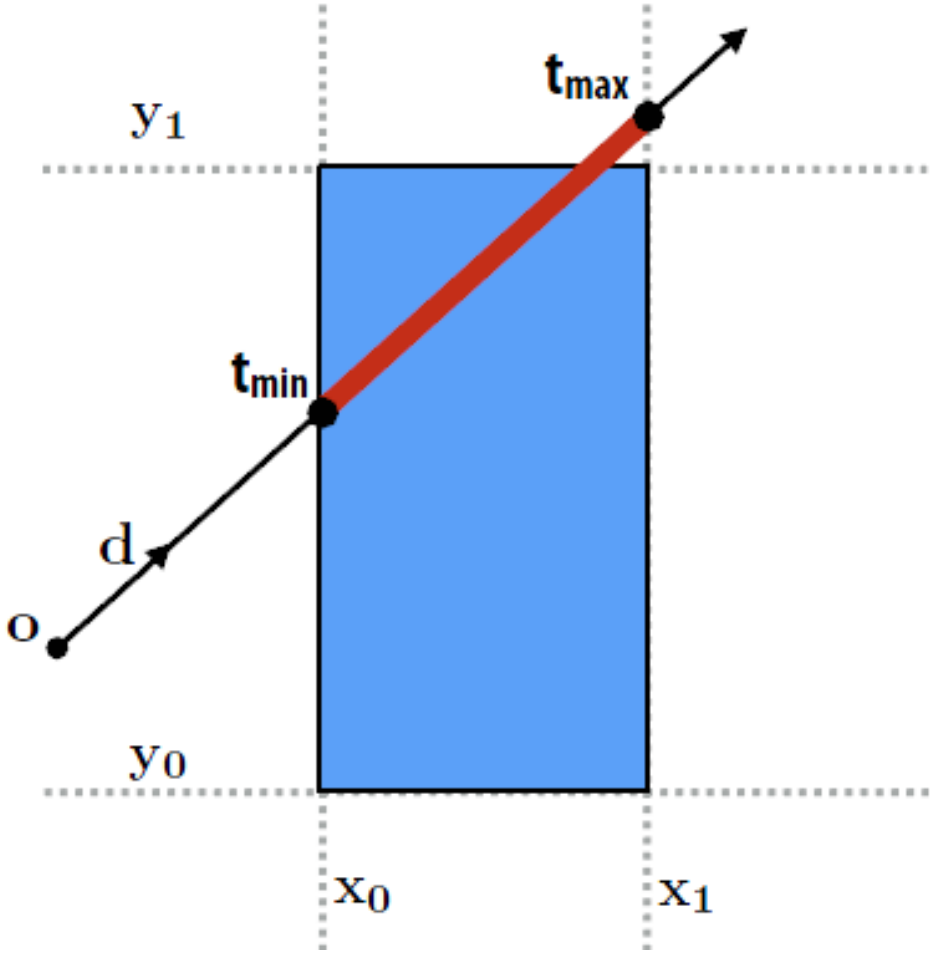


Figure shows intersections with *x*=*x*0 and *x*=*x*1 planes.

Find intersection of ray with all planes of box:

$$\mathbf{N}^{\mathbf{T}}(\mathbf{o} + t\mathbf{d}) = c$$

Math simplifies greatly since plane is axis aligned (consider x=x_0 plane in 2D):

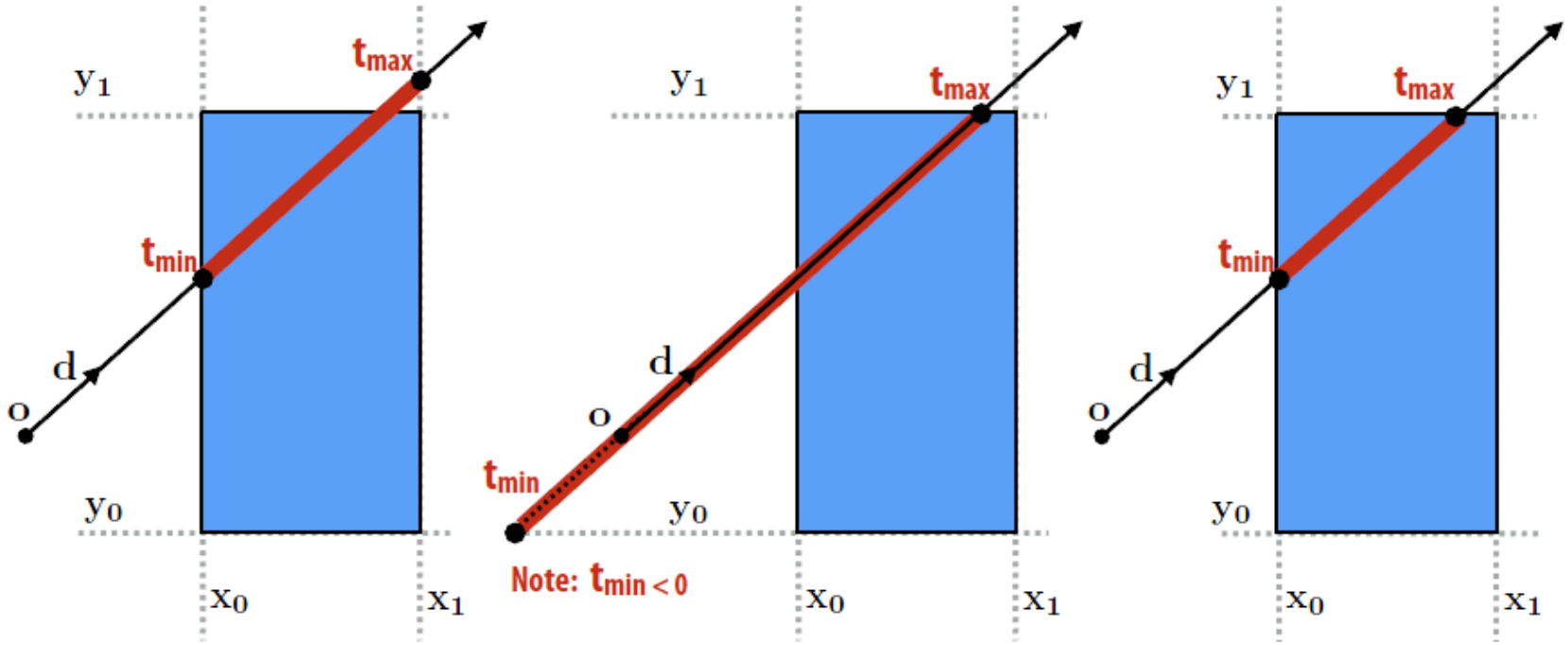
$$\mathbf{N}^{\mathbf{T}} = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$$

$$c = x_0$$

$$t = \frac{x_0 - \mathbf{o_x}}{\mathbf{d_x}}$$

Ray-axis-aligned-box intersection

- Compute intersections with all planes, take intersection of tmin/tmax intervals



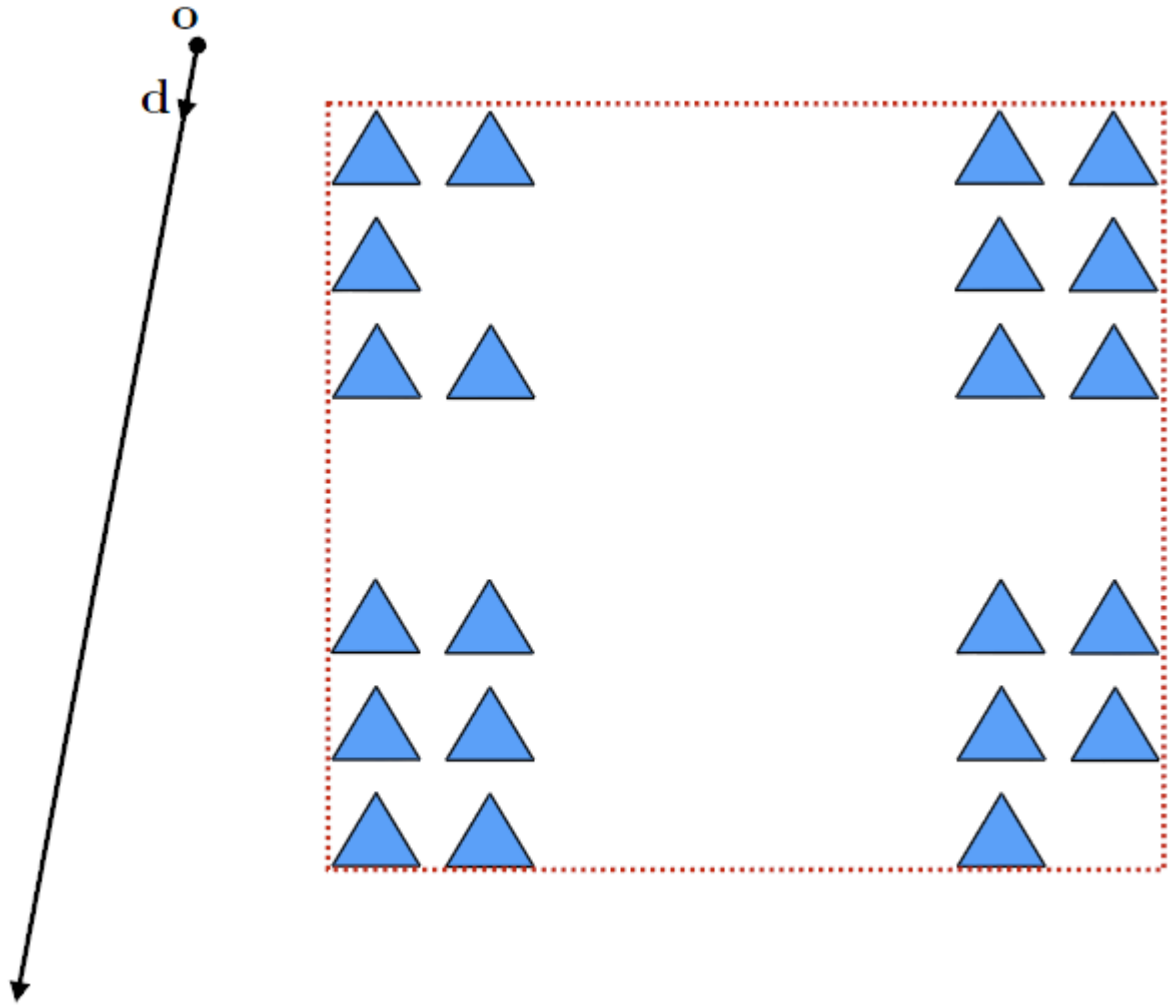
How do we know when the ray misses the box?

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

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

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

Only one bbox & Missing

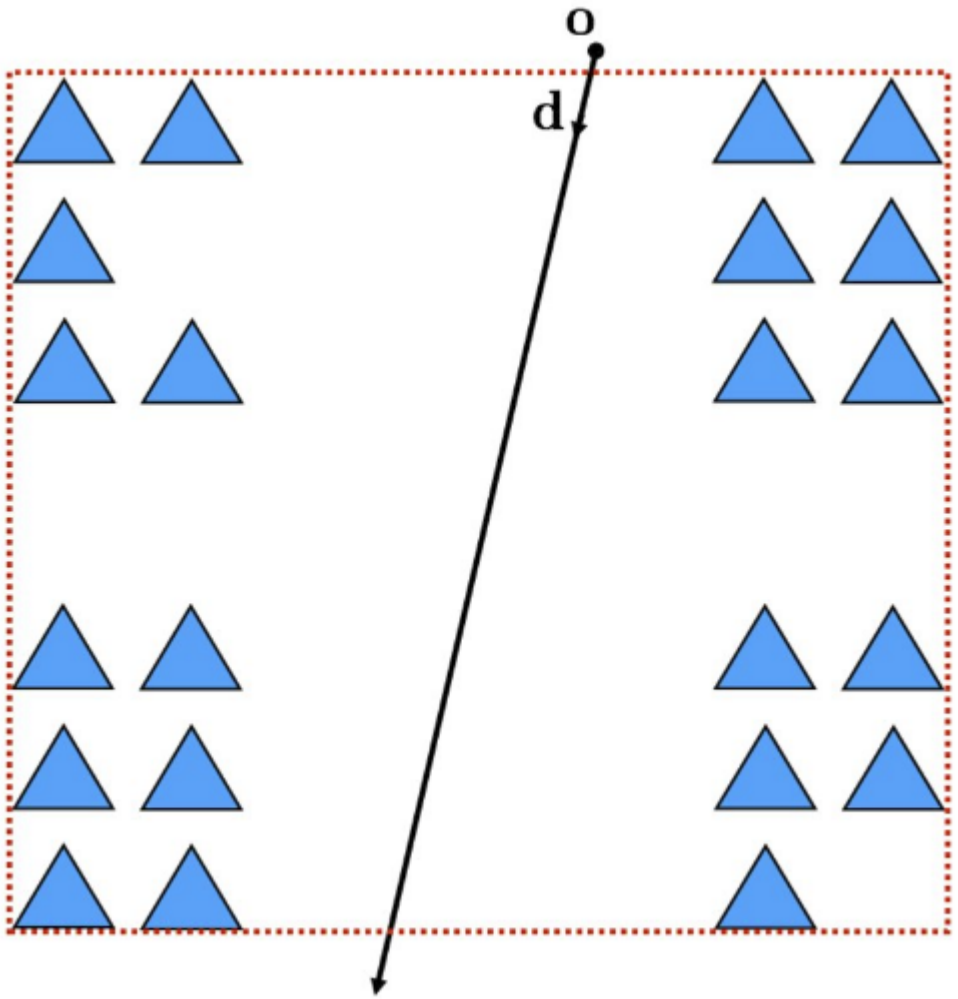


Cost (misses box):

- **preprocessing: O(n)**
- **ray-box test: O(1)**
- **amortized cost*: O(1)**

*over many ray-scene intersection tests

Only one bbox & Hitted



Cost (hits box):

- preprocessing: O(n)
- ray-box test: O(1)
- triangle tests: O(n)
- amortized cost*: O(n)

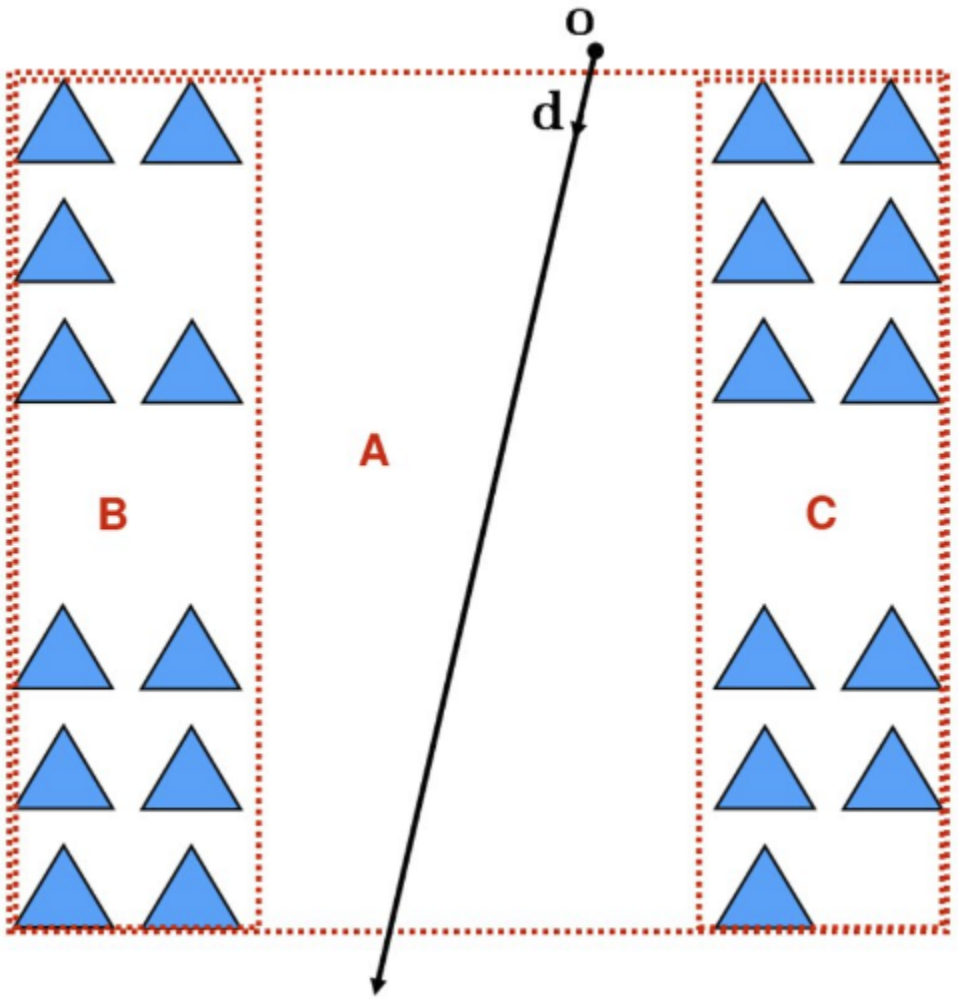
Still no better than naïve algorithm (test all triangles)!

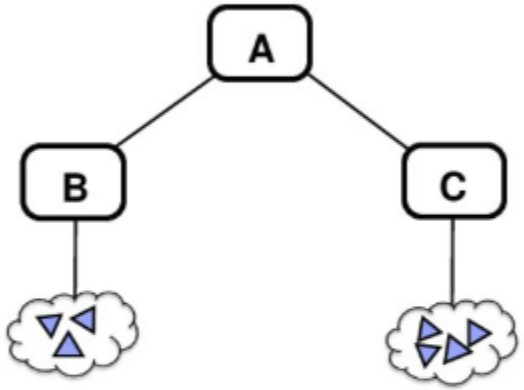
Q: How can we do better?

~~A: Use deep learning.~~

A: Apply this strategy hierarchically.

Another simple case



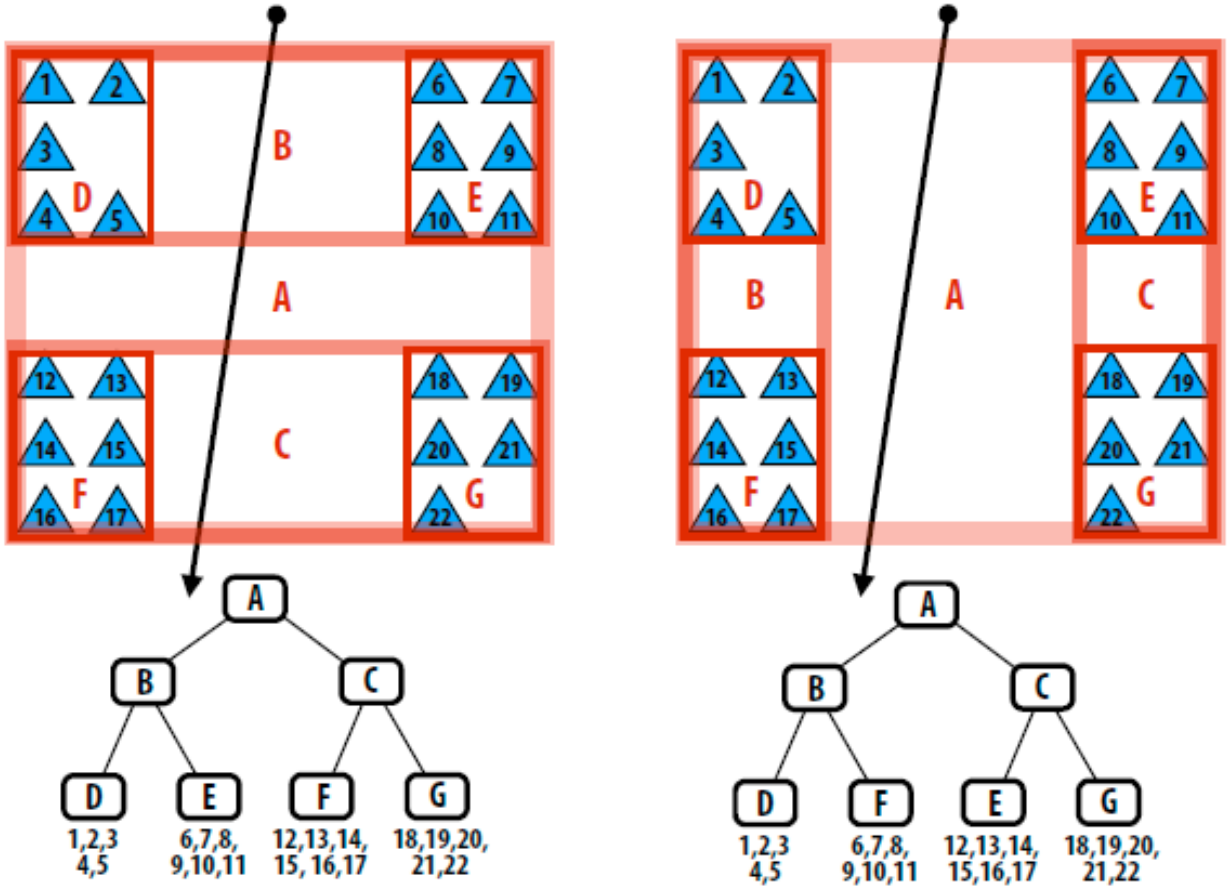


A bounding box of bounding boxes!
There is no reason to stop there!

Bounding volume hierarchy (BVH)

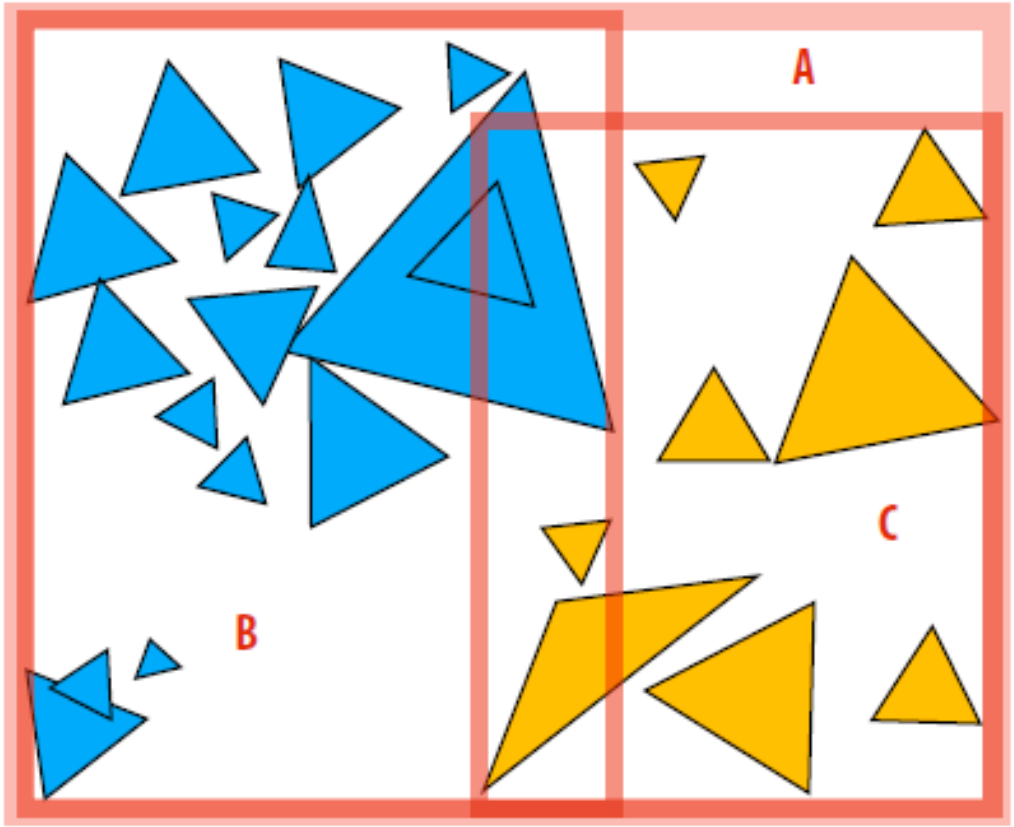
- Interior nodes:
  - **Stores subtrees**
  - Not store primitives directly
- Leaf nodes:
  - **store small list of primitives**

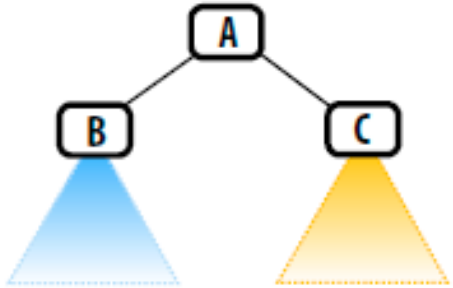
Two different BVH organizations of the same scene containing 22 primitives. Leaf node are the same.



Another BVH example

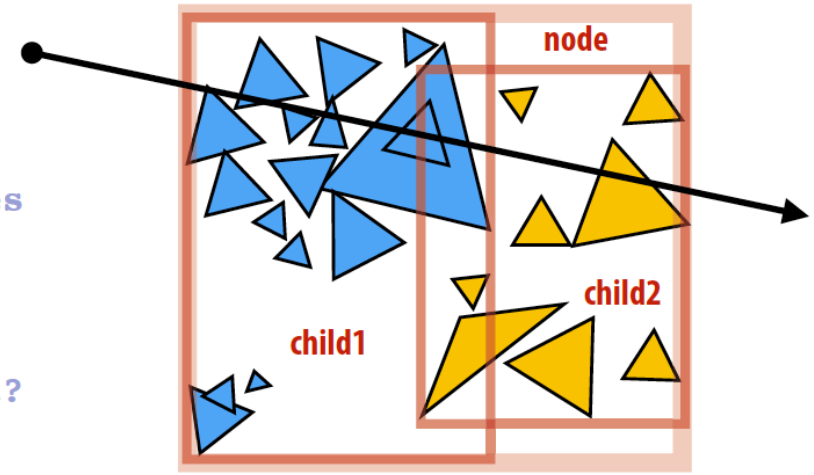
- BVH partitions each node's primitives into disjoints sets
  - Note: The sets **can still be overlapping in space** (below: child bounding boxes may overlap in space)





Ray-scene intersection using a BVH

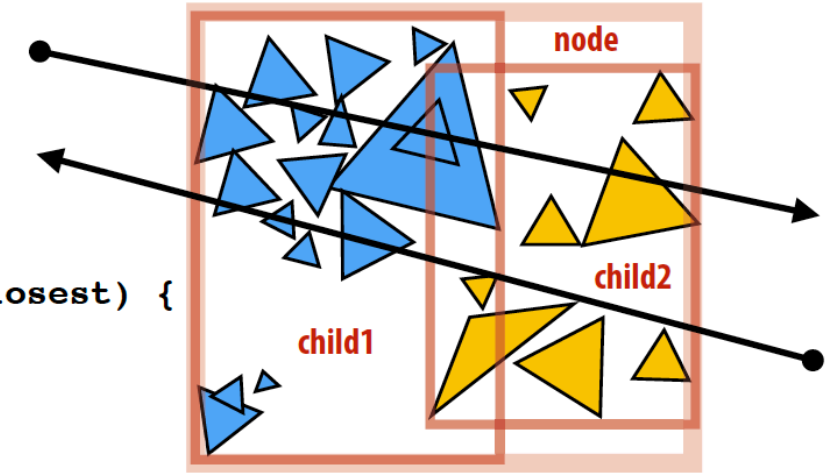
```
struct BVHNode {
   bool leaf;  // am I a leaf node?
   BBox bbox;  // min/max coords of enclosed primitives
   BVHNode* child1; // "left" child (could be NULL)
   BVHNode* child2; // "right" child (could be NULL)
   Primitive* primList; // for leaves, stores primitives
};

struct HitInfo {
   Primitive* prim;  // which primitive did the ray hit?
   float t;   // at what t value?
};

void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest) {
   HitInfo hit = intersect(ray, node->bbox);  // test ray against node's bounding box
   if (hit.prim == NULL || hit.t > closest.t))
      return; // don't update the hit record

   if (node->leaf) {
      for (each primitive p in node->primList) {
         hit = intersect(ray, p);
         if (hit.prim != NULL && hit.t < closest.t) {
            closest.prim = p;
            closest.t = t;
         }
      }
   } else {
      find_closest_hit(ray, node->child1, closest);
      find_closest_hit(ray, node->child2, closest);
   }}
```



How could this occur?

Improvement: “front-to-back” traversal

Invariant: only call find_closest_hit() if ray intersects bbox of node.




```
void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest) {

   if (node->leaf) {
      for (each primitive p in node->primList) {
         (hit, t) = intersect(ray, p);
         if (hit && t < closest.t) {
            closest.prim = p;
            closest.t = t;
         }
      }
   } else {
      HitInfo hit1 = intersect(ray, node->child1->bbox);
      HitInfo hit2 = intersect(ray, node->child2->bbox);

      NVHNode* first = (hit1.t <= hit2.t) ? child1 : child2;
      NVHNode* second = (hit2.t <= hit1.t) ? child2 : child1;

      find_closest_hit(ray, first, closest);
      if (second child's t is closer than closest.t)
         find_closest_hit(ray, second, closest); // why might we still need to do this?
   }
}
```

“Front to back” traversal. Traverse to closest child node first. Why?

Another type of query: any hit

- **Sometimes it's useful to know if the ray hits ANY primitive in the scene at all (don't care about distance to first hit)**

```
bool find_any_hit(Ray* ray, BVHNode* node) {

   if (!intersect(ray, node->bbox))
      return false;

   if (node->leaf) {
      for (each primitive p in node->primList) {
         (hit, t) = intersect(ray, p);
         if (hit)
            return true;
   } else {
     return ( find_closest_hit(ray, node->child1, closest) ||
              find_closest_hit(ray, node->child2, closest) );
   }
}
```

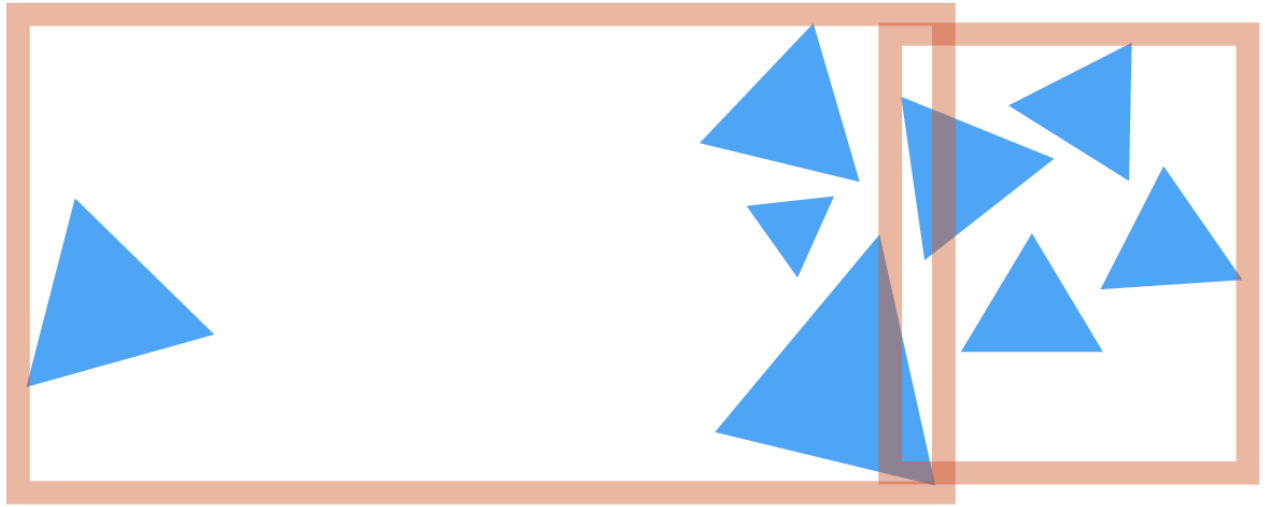
Interesting question of which child to enter first. How might you make a good decision?

For a given set of primitives, there are many possible BVHs

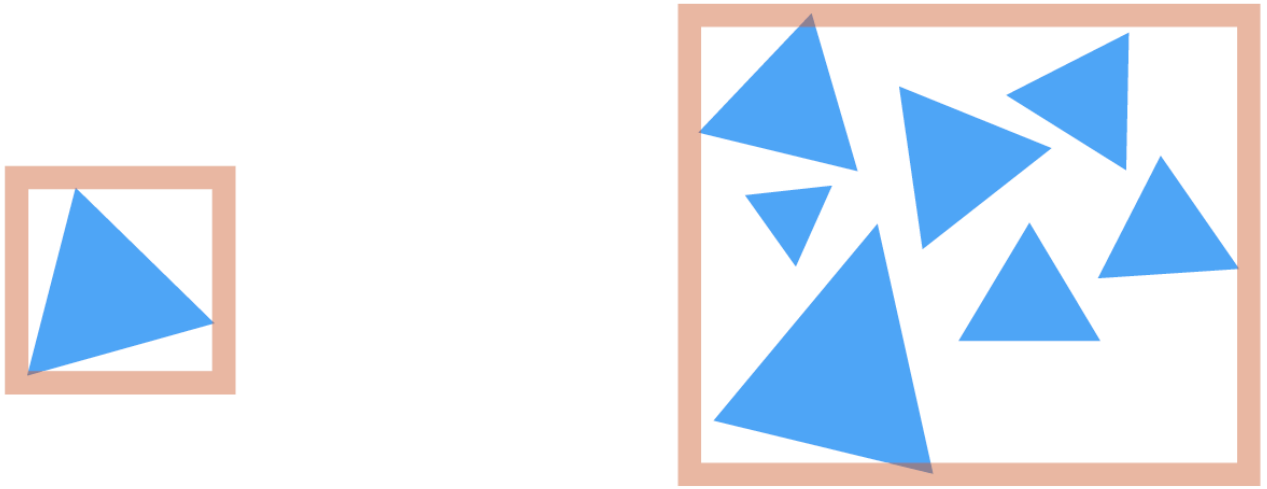
- (2N/2 ways to partition N primitives into two groups)

- Q: How do we build a high-quality BVH?

Intuition about a “good” partition?



Partition into child nodes with equal numbers of primitives



Better partition

Intuition: want small bounding boxes (minimize overlap between children, avoid empty space)

What are we really trying to do?

- A good partitioning minimizes the cost of finding the closest intersection of a ray with primitives in the node.

- If a node is a leaf node (no partitioning):

$$C = \sum_{i=1}^{N} C_{\text{isect}}(i)$$

$$= NC_{\text{isect}}$$

Where $C_{\text{isect}}(i)$ is the cost of ray-primitive intersection for primitive *i* in the node.

(Common to assume all primitives have the same cost)

Cost of making a partition

- **The expected cost of ray-node intersection, given that the node's primitives are partitioned into child sets A and B is:**

$$C = C_{\text{trav}} + p_A C_A + p_B C_B$$

C_{trav} **is the cost of traversing an interior node (e.g., load data, bbox check)**

C_A **and** C_B **are the costs of intersection with the resultant child subtrees**

p_A **and** p_B **are the probability a ray intersects the bbox of the child nodes A and B**

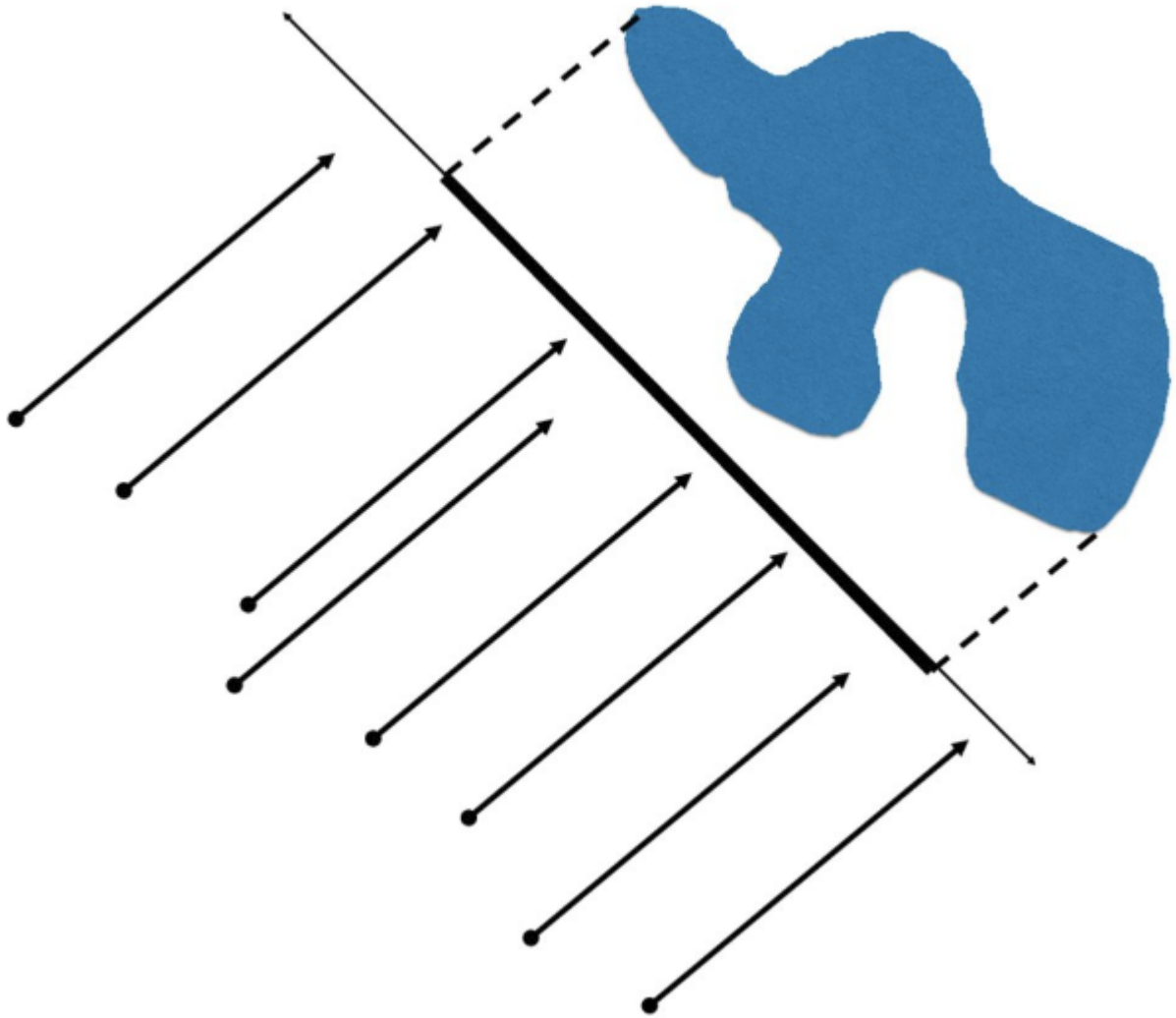
Primitive count is common approximation for child node costs:

$$C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}}$$

Where: $N_A = |A|, N_B = |B|$

Estimating probabilities

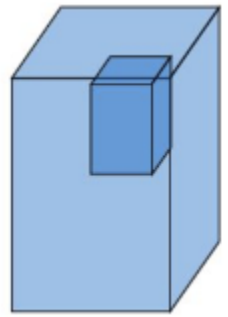
- For a given direction, the number of rays that hits an object is proportional to the projected area



Estimating probabilities

- **For convex object A inside convex object B, the probability that a random ray that hits B also hits A is given by the ratio of the surface areas SA and SB of these objects.**

$$P(\text{hit}A|\text{hit}B) = \frac{S_A}{S_B}$$



- **Surface area heuristic (SAH):**

$$C = C_{\text{trav}} + \frac{S_A}{S_N} N_A C_{\text{isect}} + \frac{S_B}{S_N} N_B C_{\text{isect}}$$

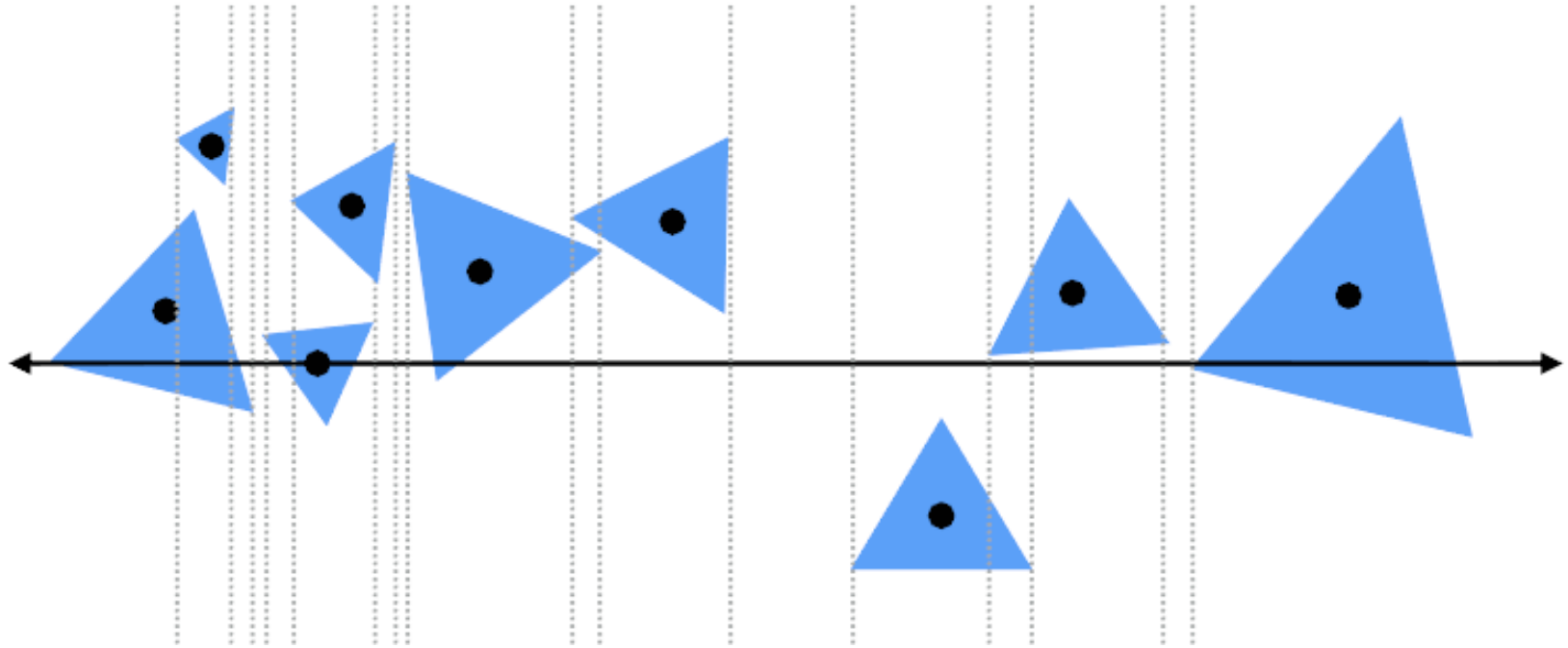
Assumptions of the SAH (may not hold in practice):

Rays are randomly distributed

Rays are not occluded

Implementing partitions for build BVH

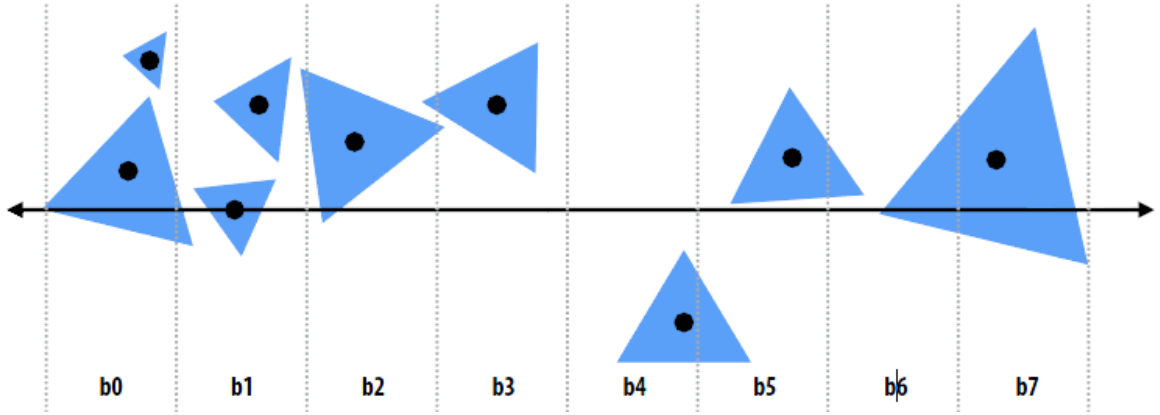
- Constrain search for good partitions to axis-aligned spatial partitions
  - Choose an axis; Choose a split plane on that axis
  - Partition primitives by the side of splitting plane their centroid lies
  - SAH changes only when split plane moves past triangle boundary
  - Have to consider rather large number of possible split planes…



Build BVH

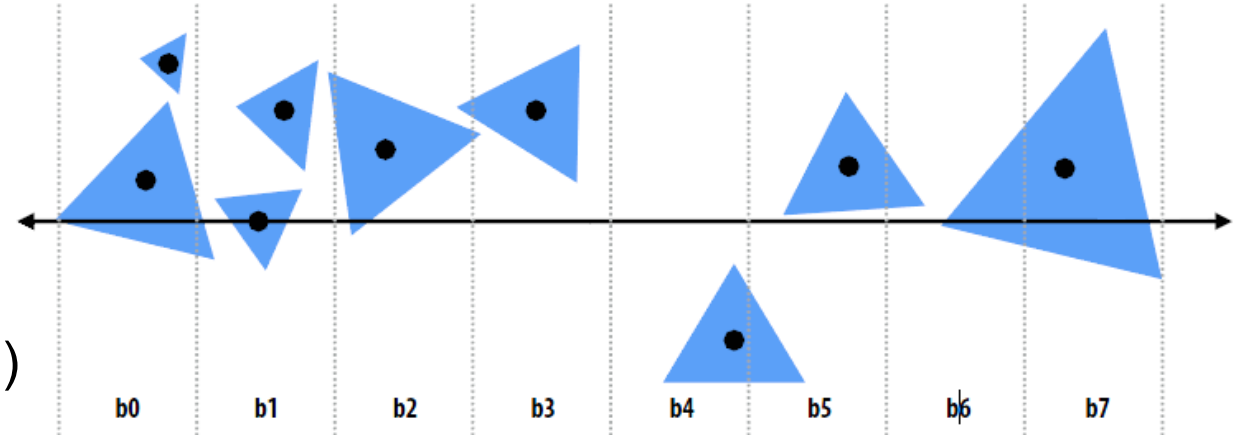
x = 0, y = 1, and z = 2

```
function bvh-node::create(object-array A, int AXIS)
    N = A.length
    if (N= 1) then
        left = A[0]
        right = NULL
        bbox = bounding-box(A[0])
    else if (N= 2) then
        left-node = A[0]
        right-node = A[1]
        bbox = combine(bounding-box(A[0]), bounding-box(A[1]))
    else
        sort A by the object center along AXIS
        left= new bvh-node(A[0..N/2 − 1], (AXIS +1) mod 3)
        right = new bvh-node(A[N/2..N−1], (AXIS +1) mod 3)
        bbox = combine(left → bbox, right → bbox)
```



Efficiently building BVH

- Efficient modern approximation: split spatial extent of primitives into B buckets (B is typically small: B < 32)



$$C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}}$$

function bvh-node::create(object-array A, int AXIS)
if (N==1) then … else if (N==2) …
else

initialize buckets along AXIS

For each primitive p in A:

b = compute_bucket(p.centroid)

b.bbox.union(p.bbox);

b.prim_count++;

For each of the B-1 possible partitioning planes evaluate SAH, return array A1, A2

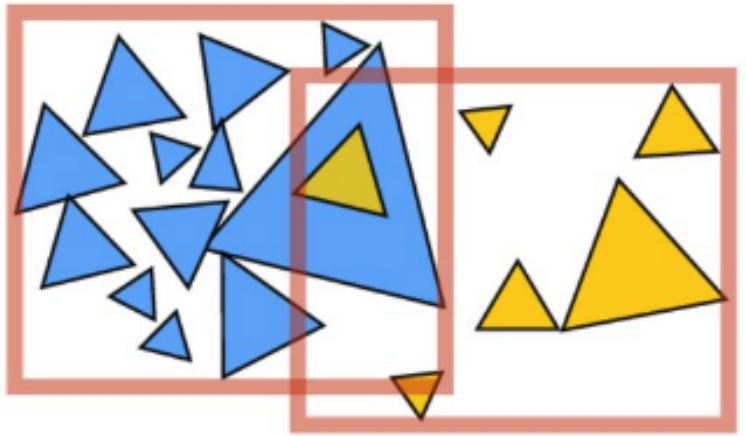
left= new bvh-node(A1, (AXIS +1) mod 3)

right = new bvh-node(A2, (AXIS +1) mod 3)

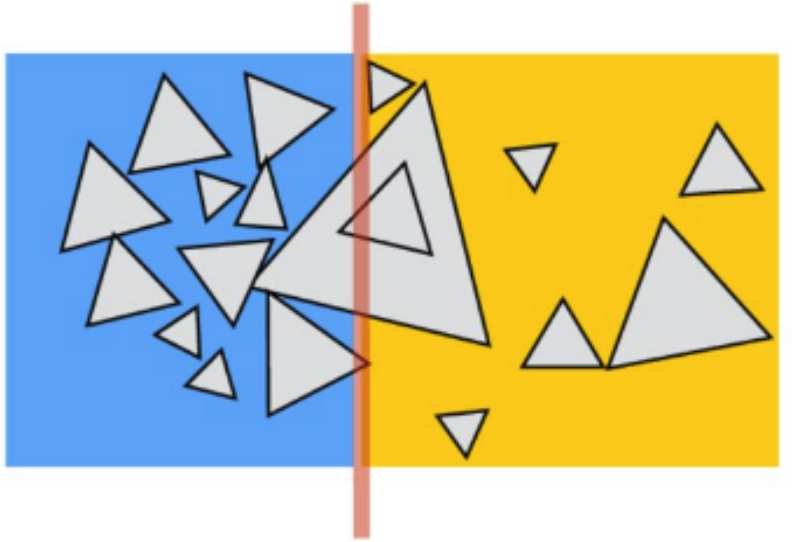
bbox = combine(left→bbox, right→bbox)

Primitive-partitioning acceleration structures vs. space-partitioning structures

- **Primitive partitioning** (BVH): partitions node's primitives into disjoint sets (but sets may overlap in space)

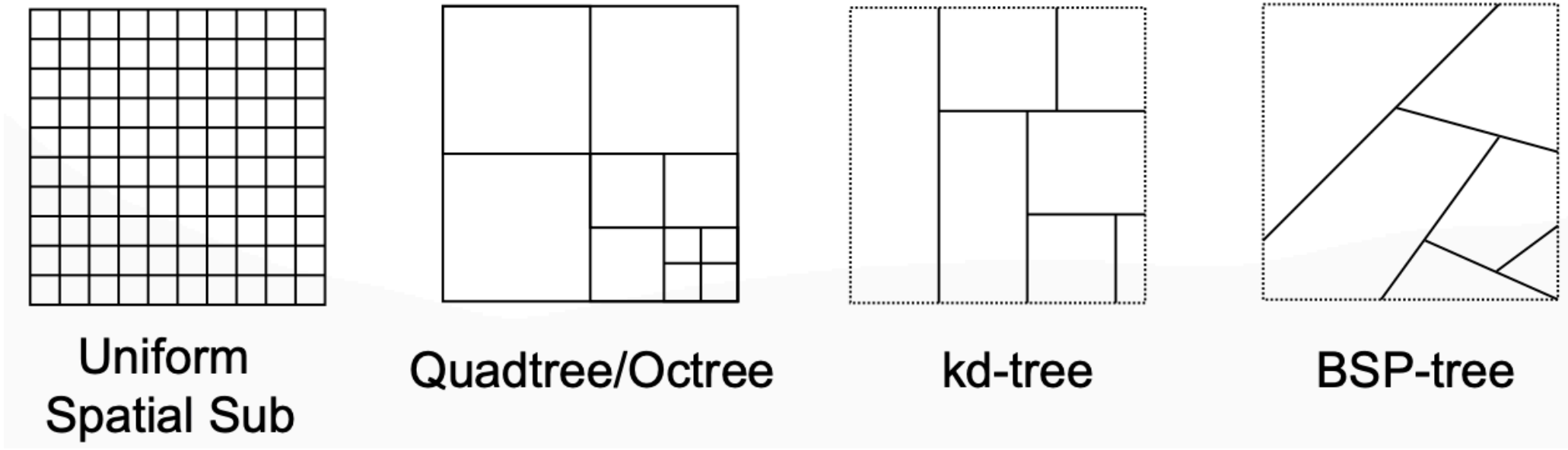


- **Space-partitioning** (grid, K-D tree) partitions space into disjoint regions (primitives may be contained in multiple regions of space)



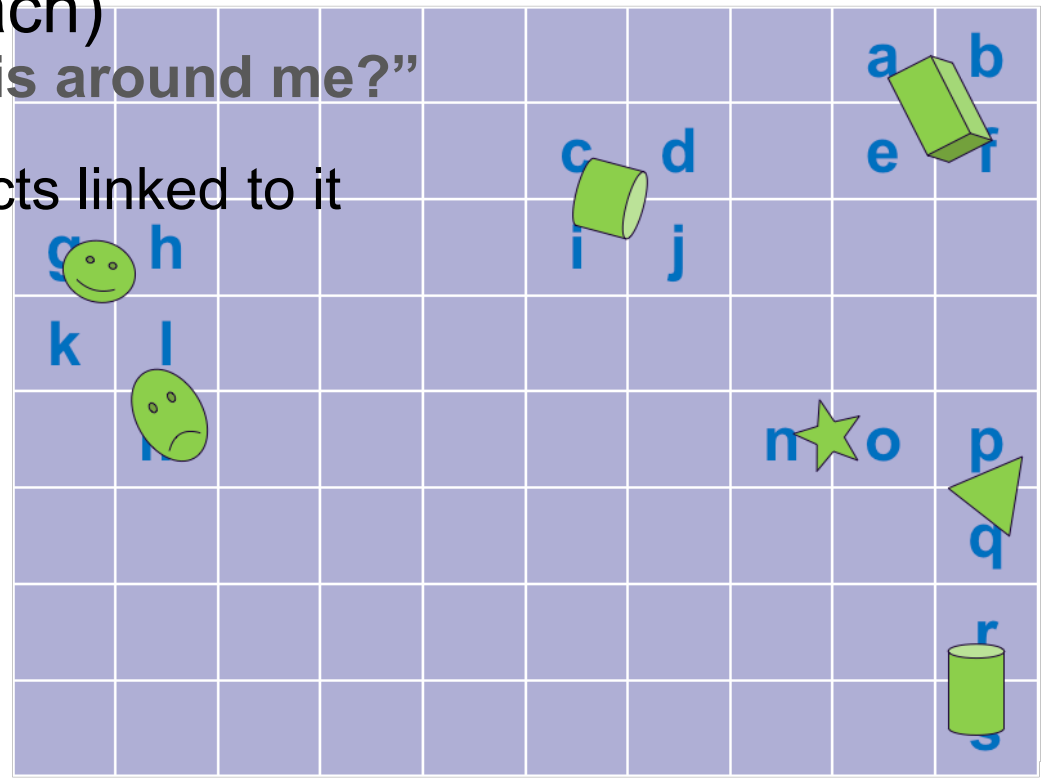
Space-partitioning

- Basic techniques:



Uniform grid

- Partition space into equal sized volumes (voxel/cell)
- Each grid cell contains **primitives that overlap voxel**. (very cheap to construct acceleration structure)
- **Indexing function**:
  - Point3D → cell index, (constant time!)
- Queries: (“gather” approach)
  - **“I’m here. Which object is around me?”**
  - given a point to test *p*,
    find cell C[j], test all objects linked to it



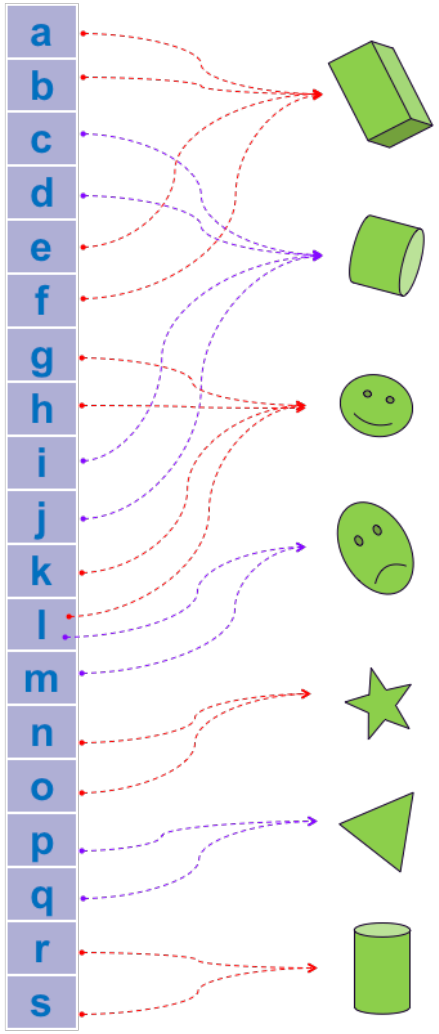
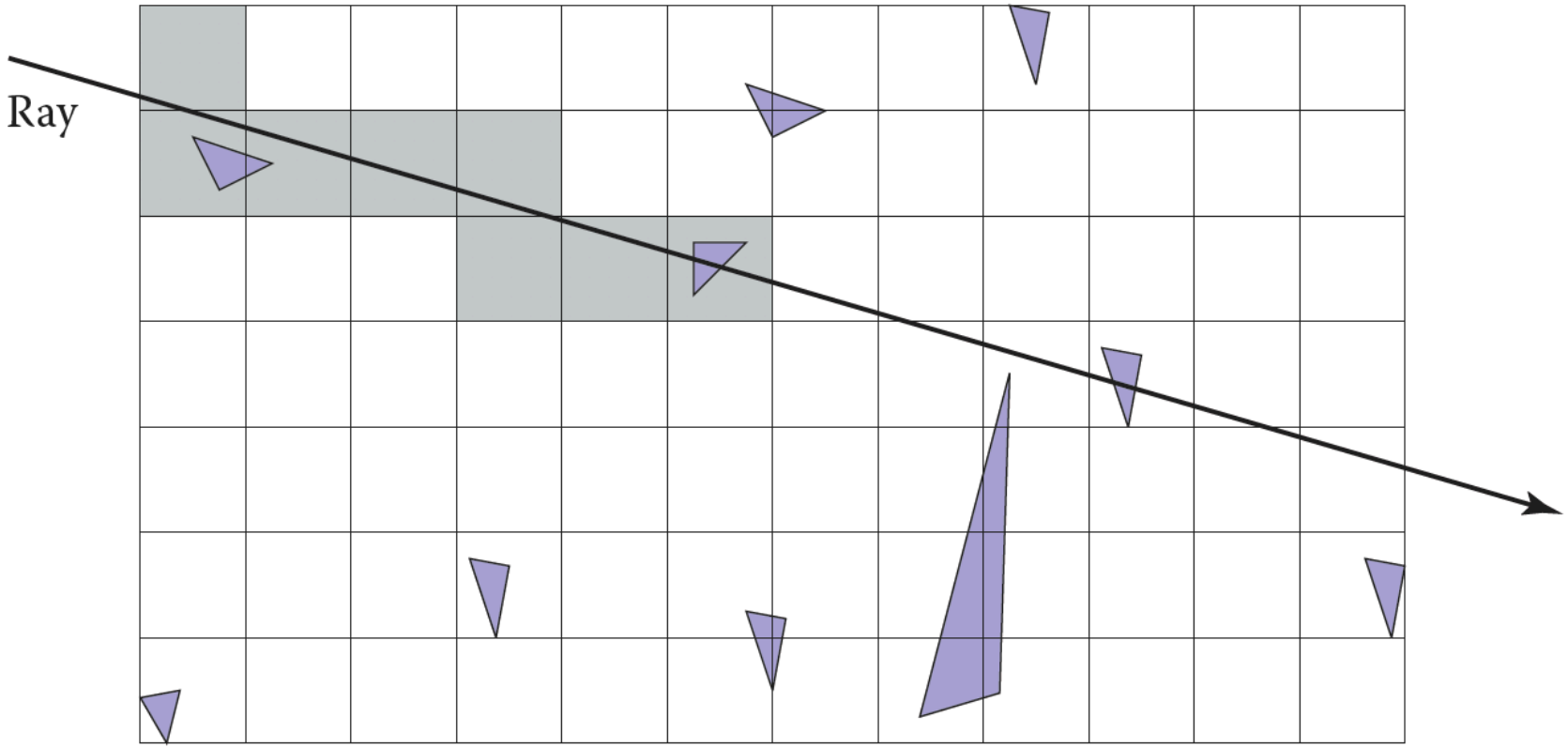


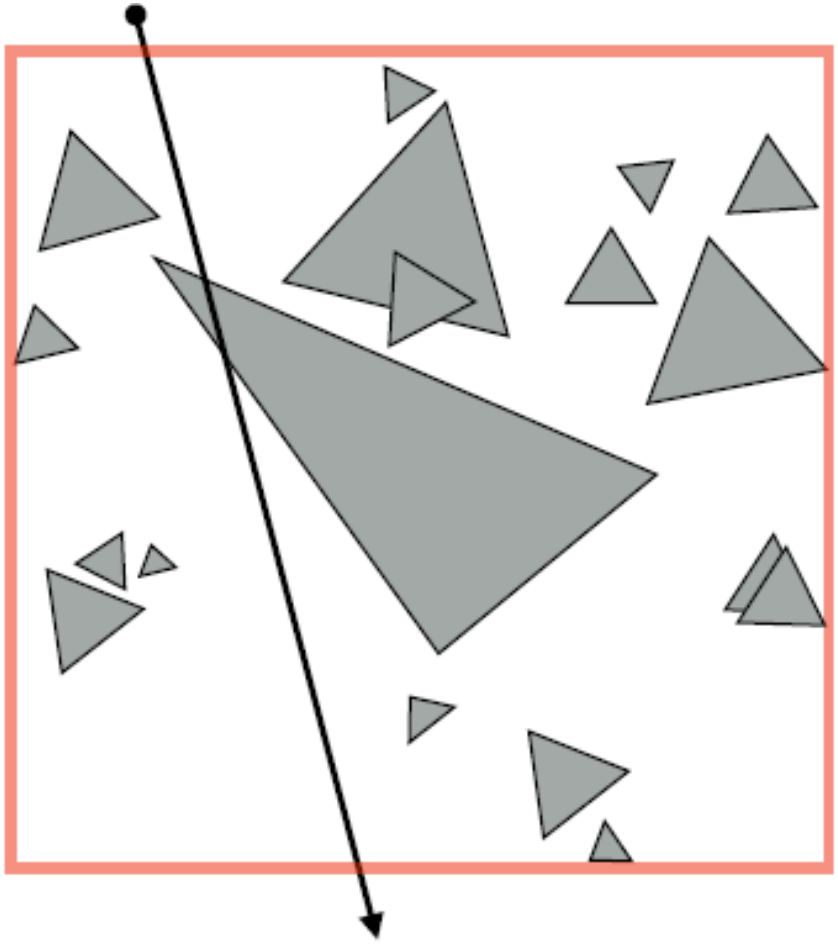
Image Copyright: Marco Tarini

Uniform grid

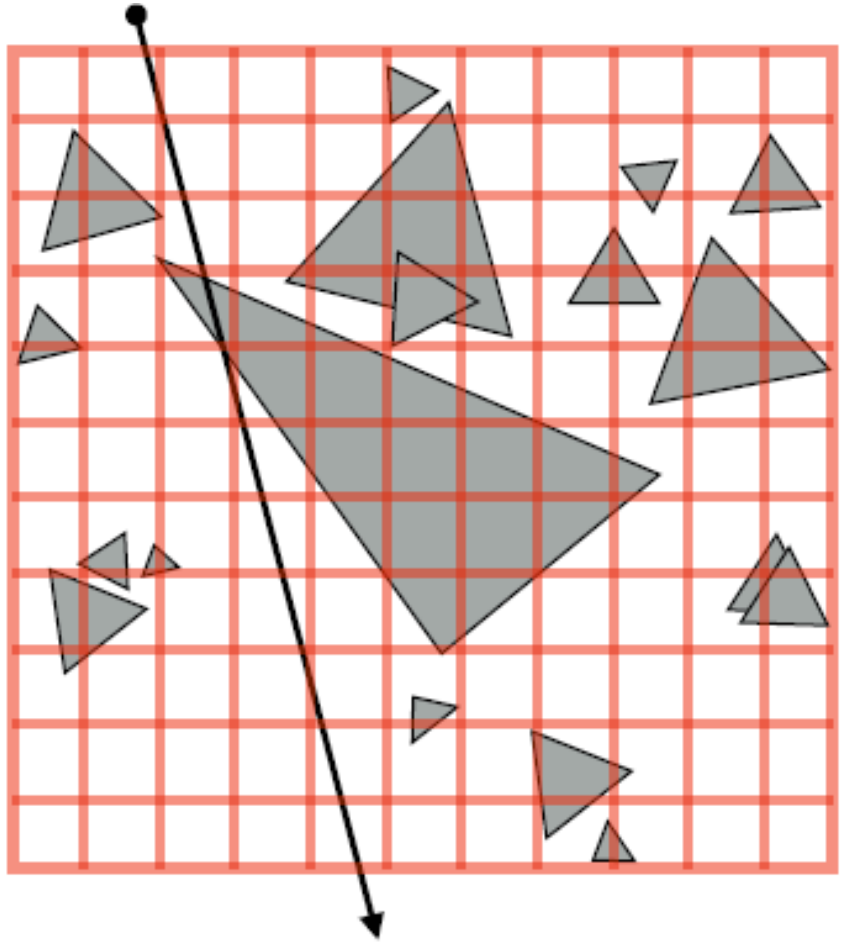
- Walk ray through volume in order
  - Very efficient implementation possible (think: **3D line rasterization**)
  - Only consider intersection with primitives in voxels the ray intersects
  - When an object is hit, the traversal ends.



What should the grid resolution be?



Too few grids cell: degenerates to brute-force approach

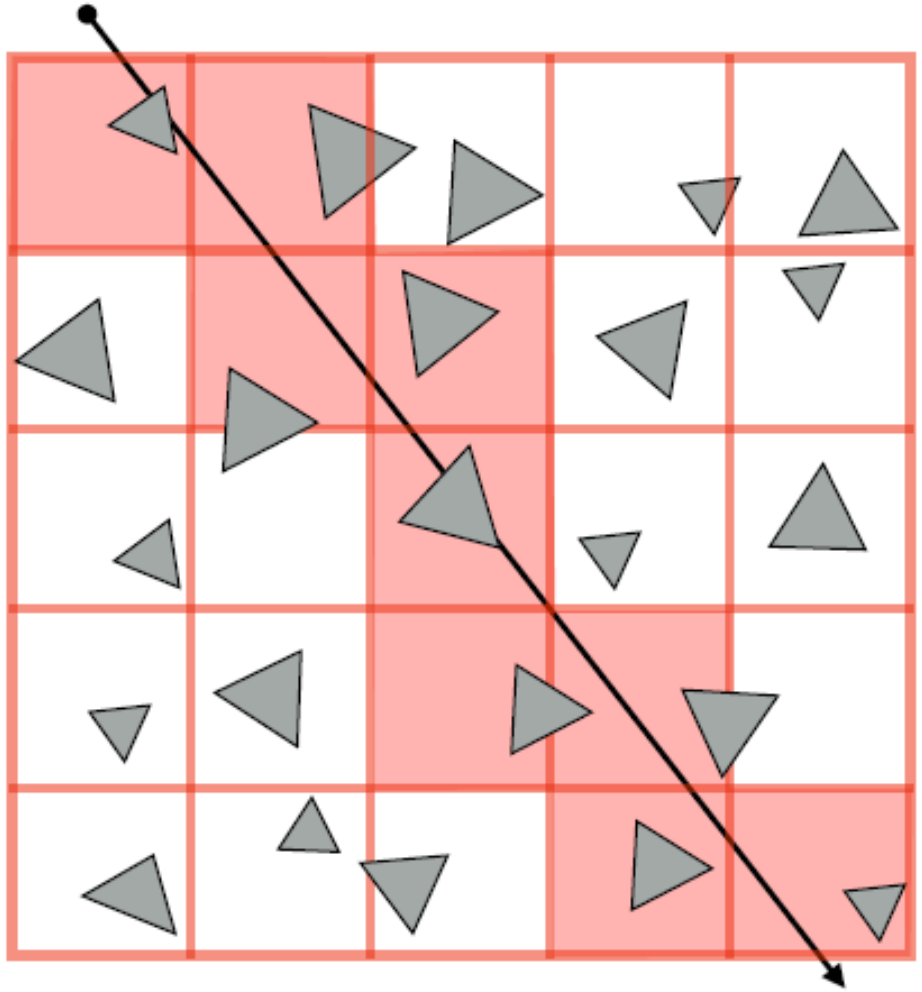


Too many grid cells: incur significant cost traversing through cells with empty space

Memory quadratic with inverse of cell size!

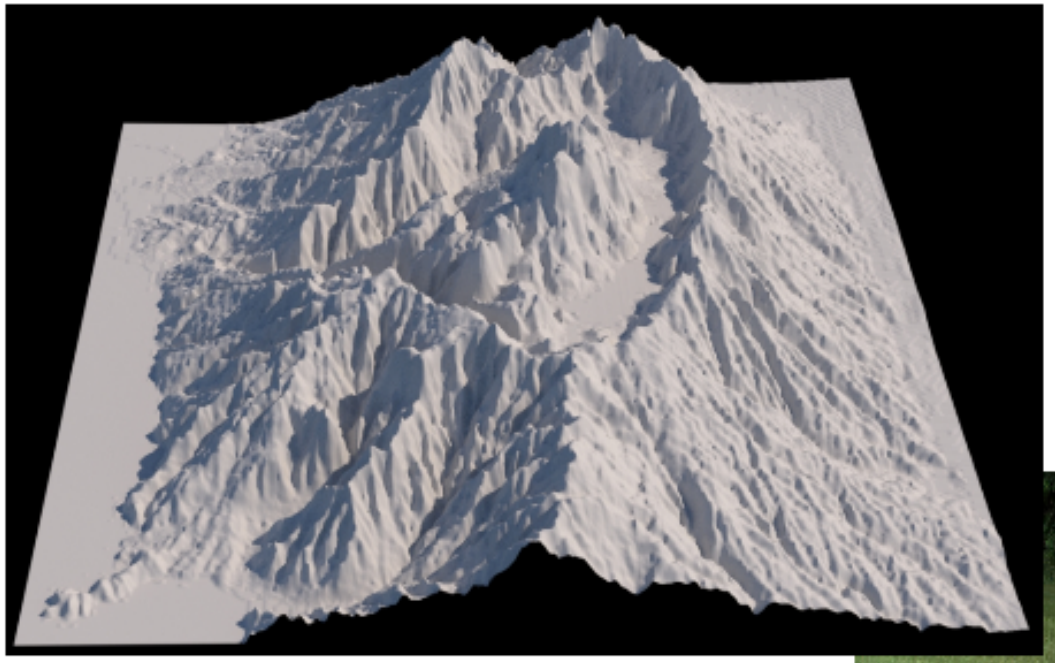
Heuristic

- Choose number of voxels ~ total number of primitives (constant prims per voxel — assuming uniform distribution of primitives)



Intersection cost: $O(\sqrt[3]{N})$

Uniform distribution of primitives



Terrain / height fields:

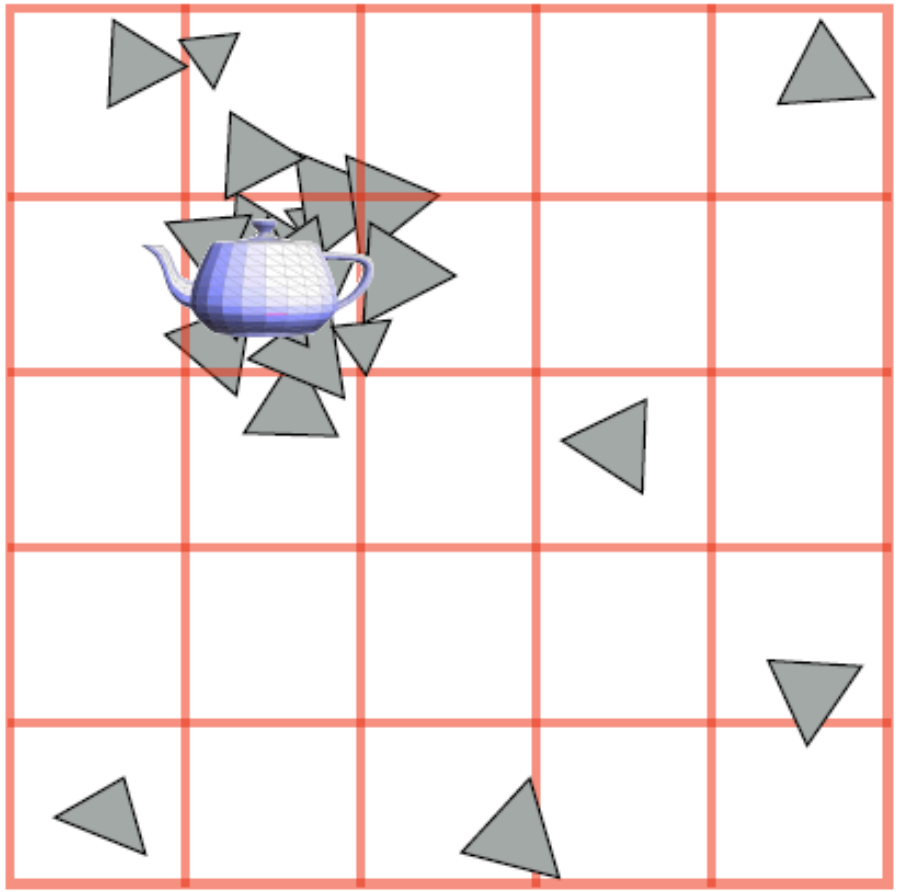
[Image credit: Misuba Renderer]

Grass:

[Image credit: www.kevinboulanger.net/grass.html]

Uniform grid cannot adapt to non-uniform distribution of geometry in scene

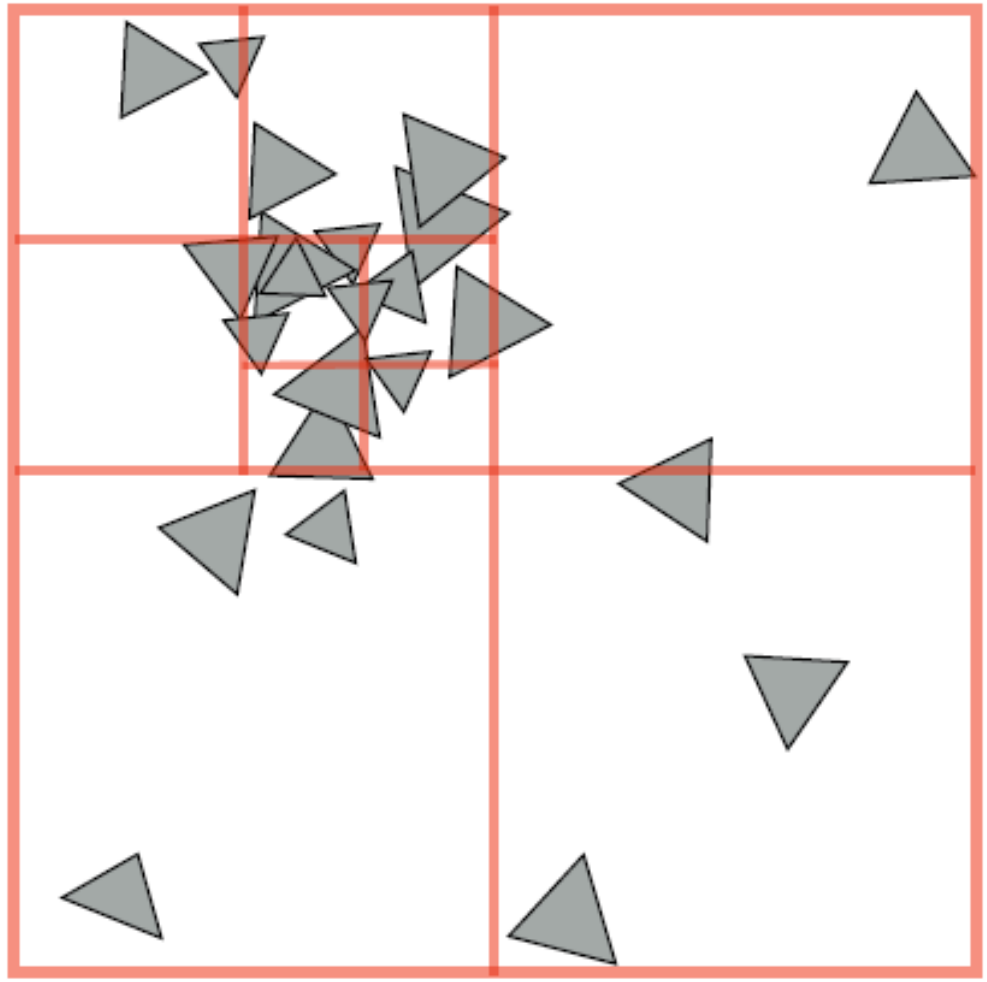
- “Teapot in a stadium problem”
- **Scene** has **large spatial extent**.
- Contains a high-resolution **object** that has **small spatial extent** (ends up in one grid cell)



Non-uniform distribution of geometric detail requires adaptive grids

Quad-tree / octree

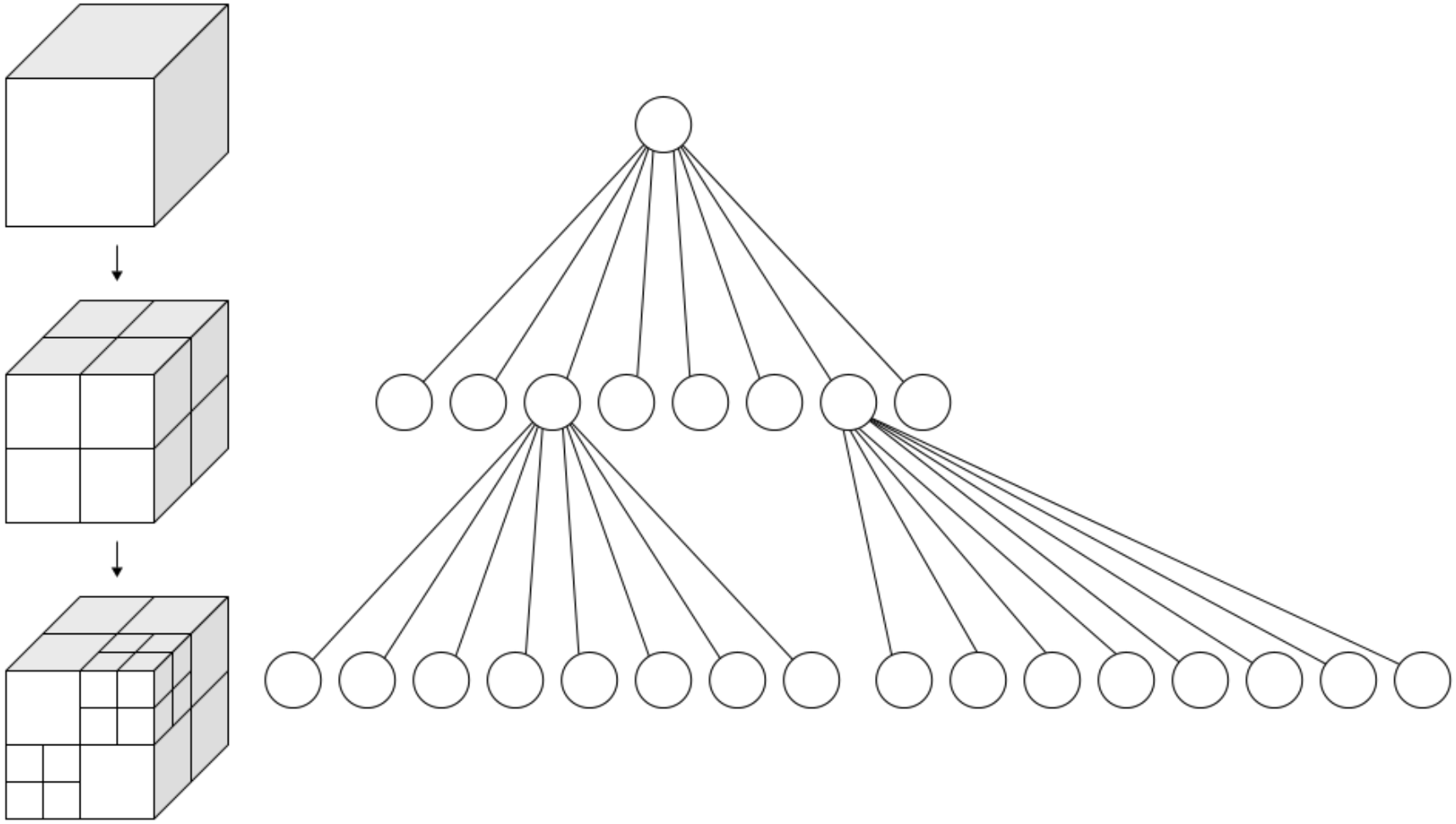
- Like uniform grid: easy to build (don’t have to choose partition planes)
- Has greater ability to adapt to location of scene geometry than uniform grid.
- But **lower intersection performance than K-D tree** (only limited ability to adapt)



Quad-tree: nodes have 4 children (partitions 2D space)

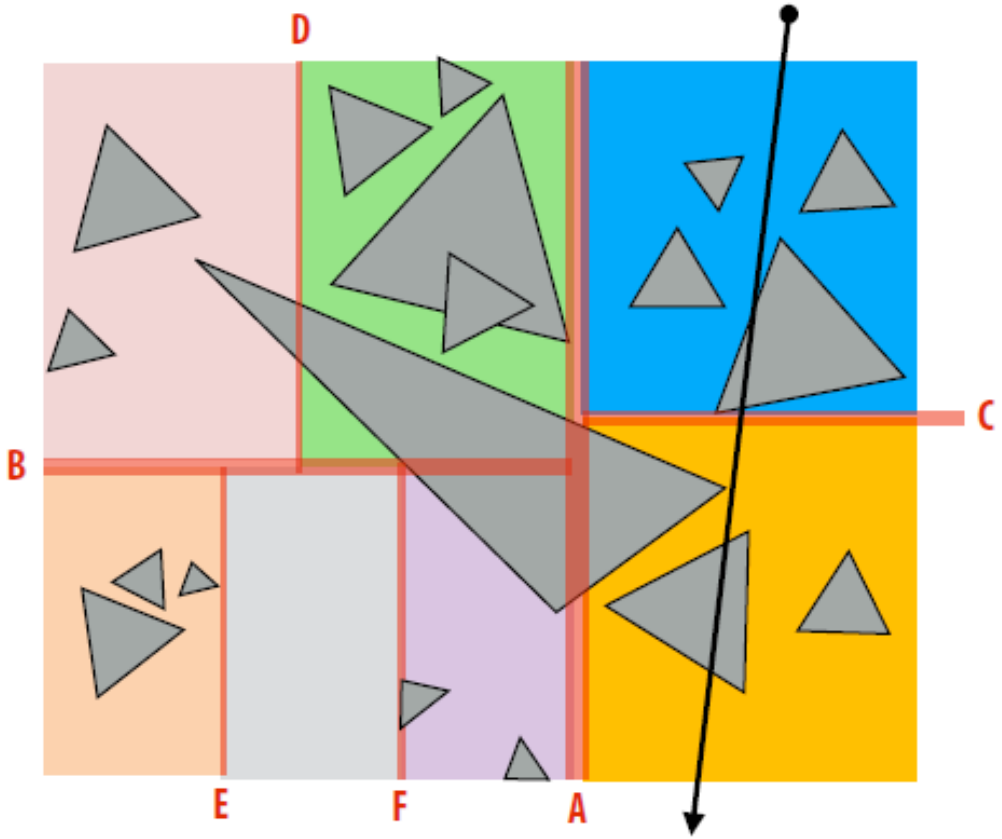
Octree: nodes have 8 children (partitions 3D space)

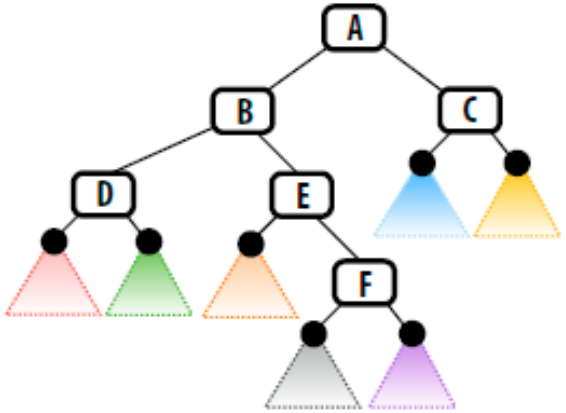
Oc-Tree (3D)



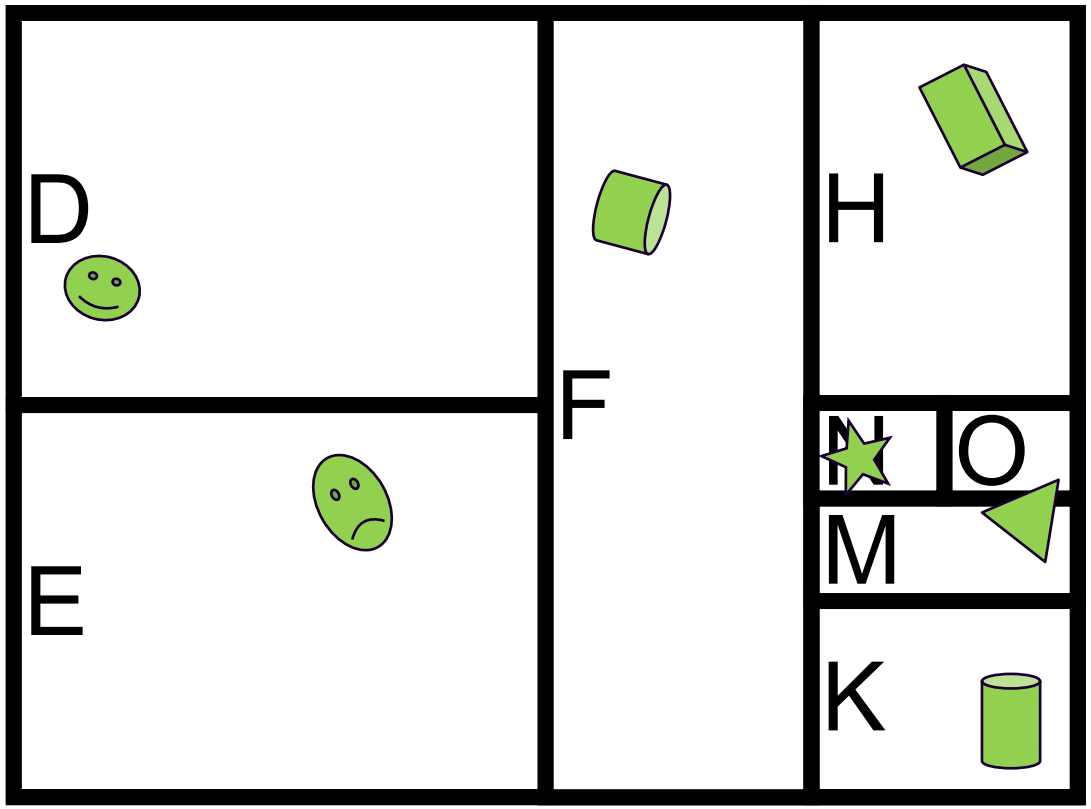
K-D tree

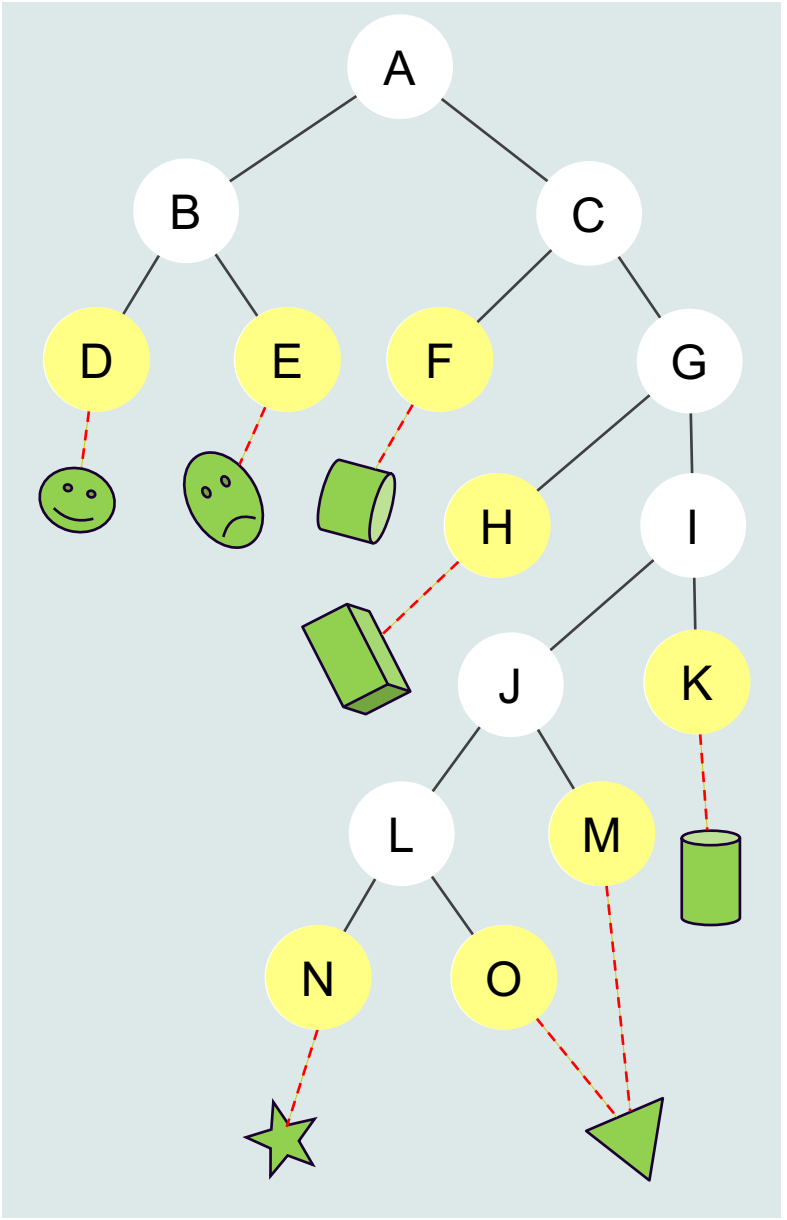
- Recursively partition space via **axis-aligned partitioning planes**
  - Interior nodes correspond to spatial splits (still correspond to spatial volume)
  - Node traversal can proceed in front-to-back order
  - unlike BVH, can **terminate search after first hit is found**





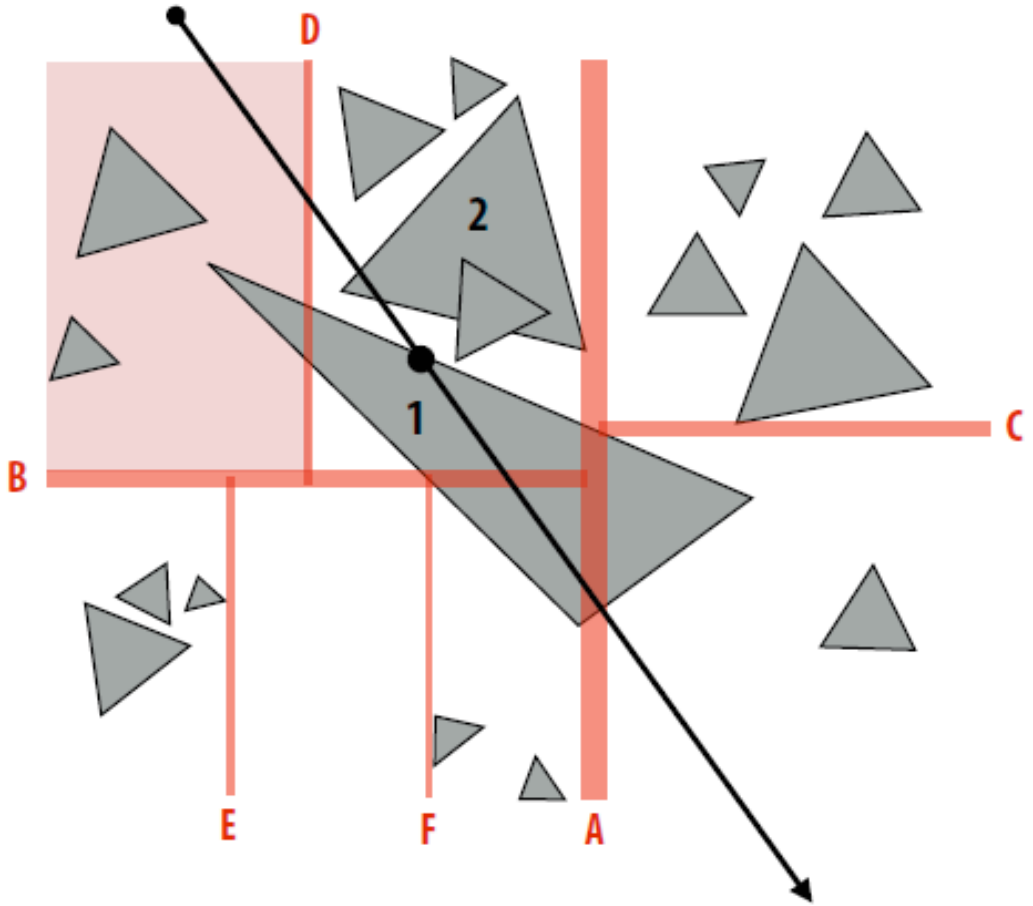
kD-tree

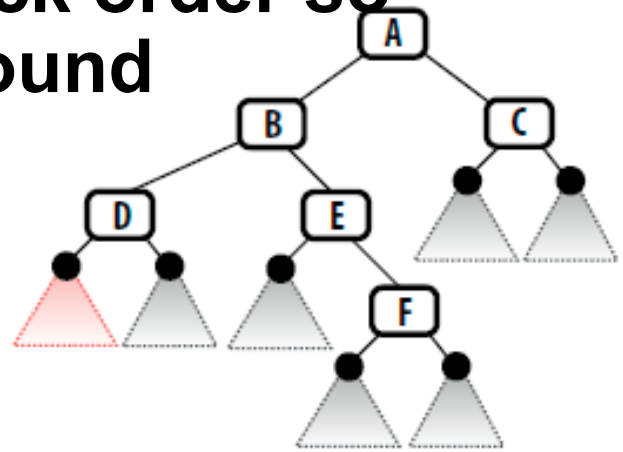




Challenge: objects overlap multiple nodes

- **Want node traversal to proceed in front-to-back order so traversal can terminate search after first hit found**





Triangle 1 overlaps multiple nodes.

Ray hits triangle 1 when in highlighted leaf cell.

But intersection with triangle 2 is closer! (Haven't traversed to that node yet)

Solution: require primitive intersection point to be within current leaf node.

(primitives may be intersected multiple times by same ray *)

Intersection of ray & node

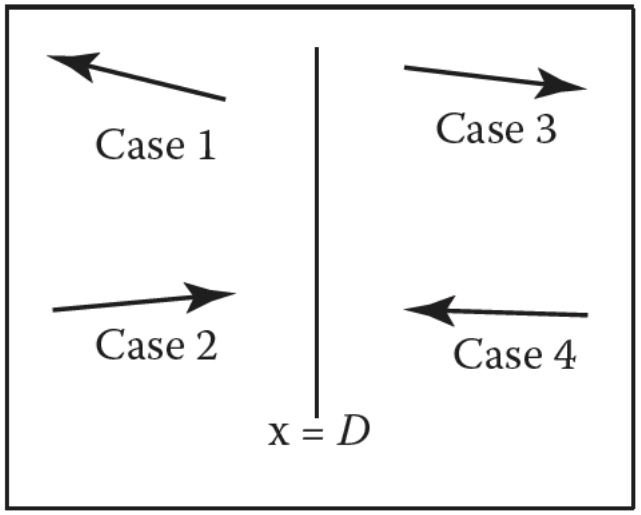
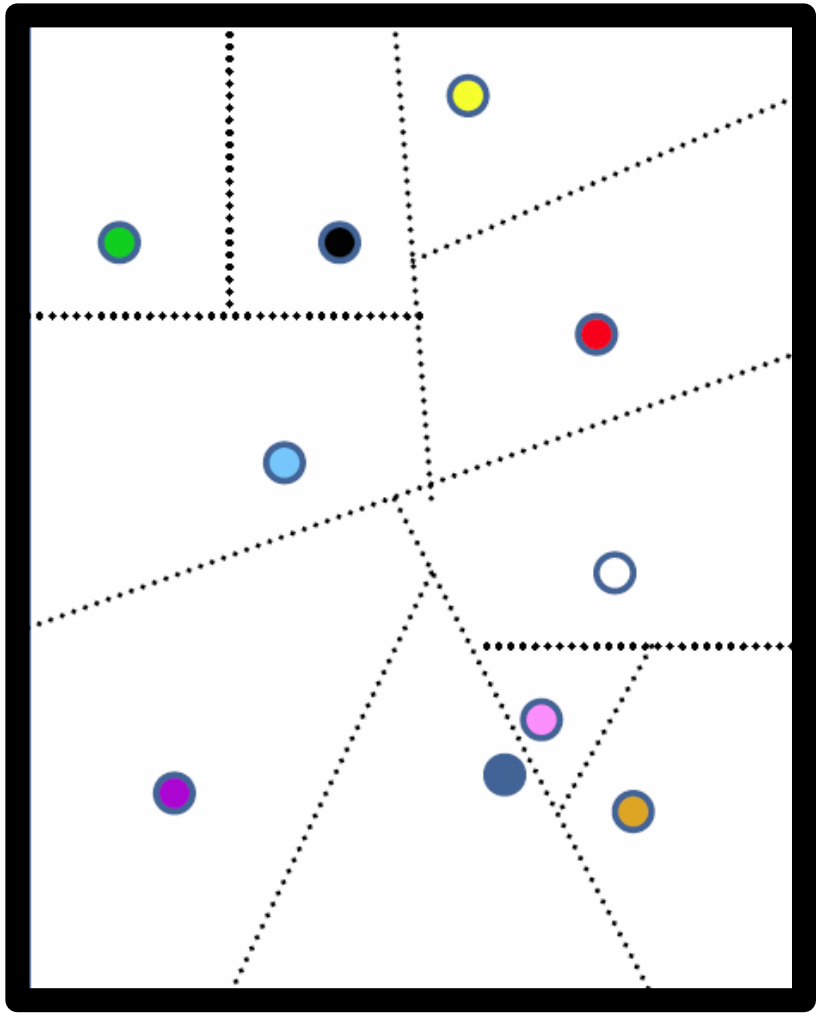


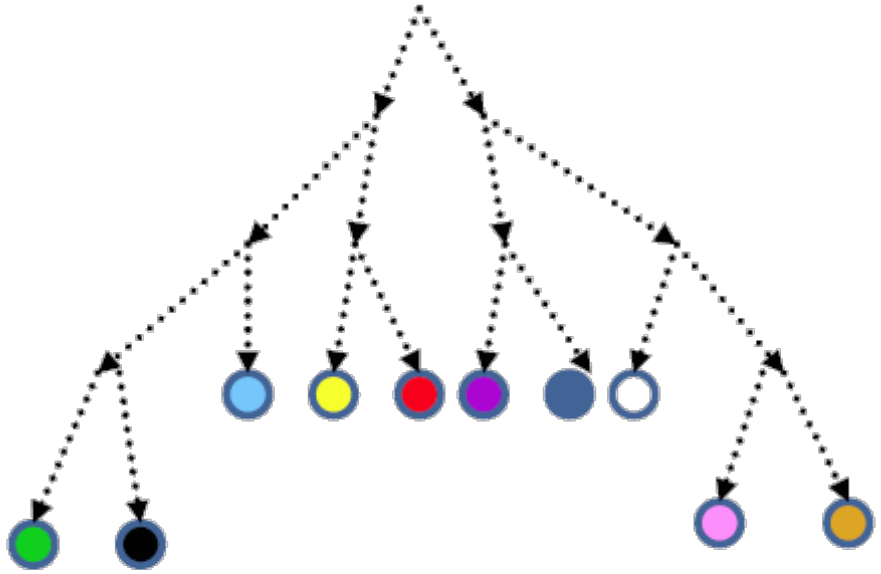
Figure 12.32. The four cases of how a ray relates to the BSP cutting plane $x = D$.

function bool bsp-node::hit(ray $\mathbf{a} + t\mathbf{b}$, real t_0, real t_1, hit-record rec)

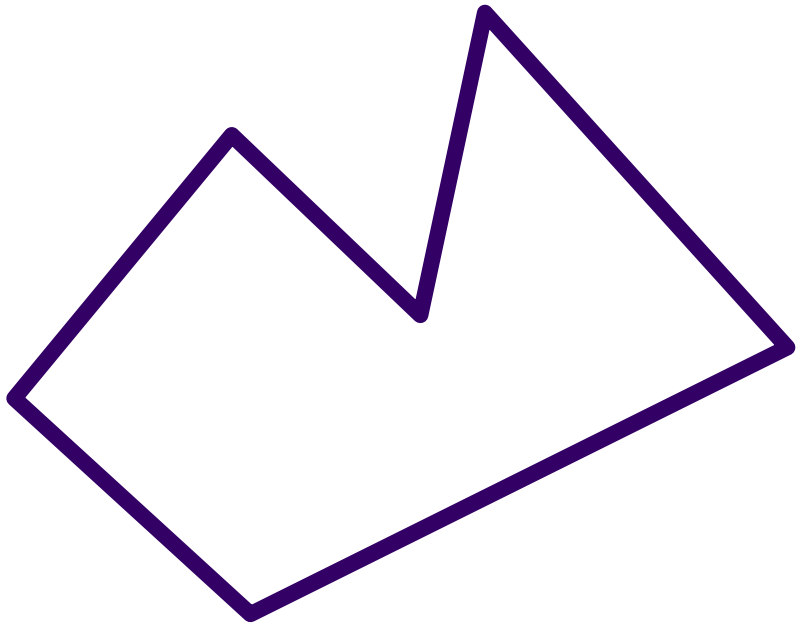
$x_p = x_a + t_0 x_b$
if $(x_p < D)$ **then**
  if $(x_b < 0)$ **then**
    return (left $\neq$ NULL) and (left$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, t_0, t_1, rec))
  $t = (D - x_a)/x_b$
  if $(t > t_1)$ **then**
    return (left $\neq$ NULL) and (left$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, t_0, t_1, rec))
  if (left $\neq$ NULL) and (left$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, t_0, t, rec)) **then**
    return true
  return (right $\neq$ NULL) and (right$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, t, t_1, rec))
else
  analogous code for cases 3 and 4

BSP-tree
Binary Spatial Partitioning tree



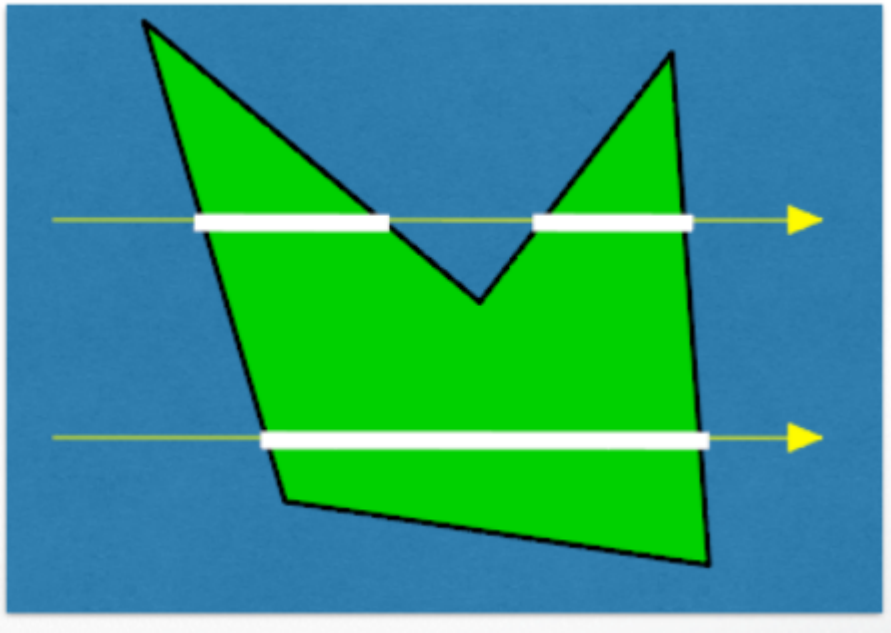


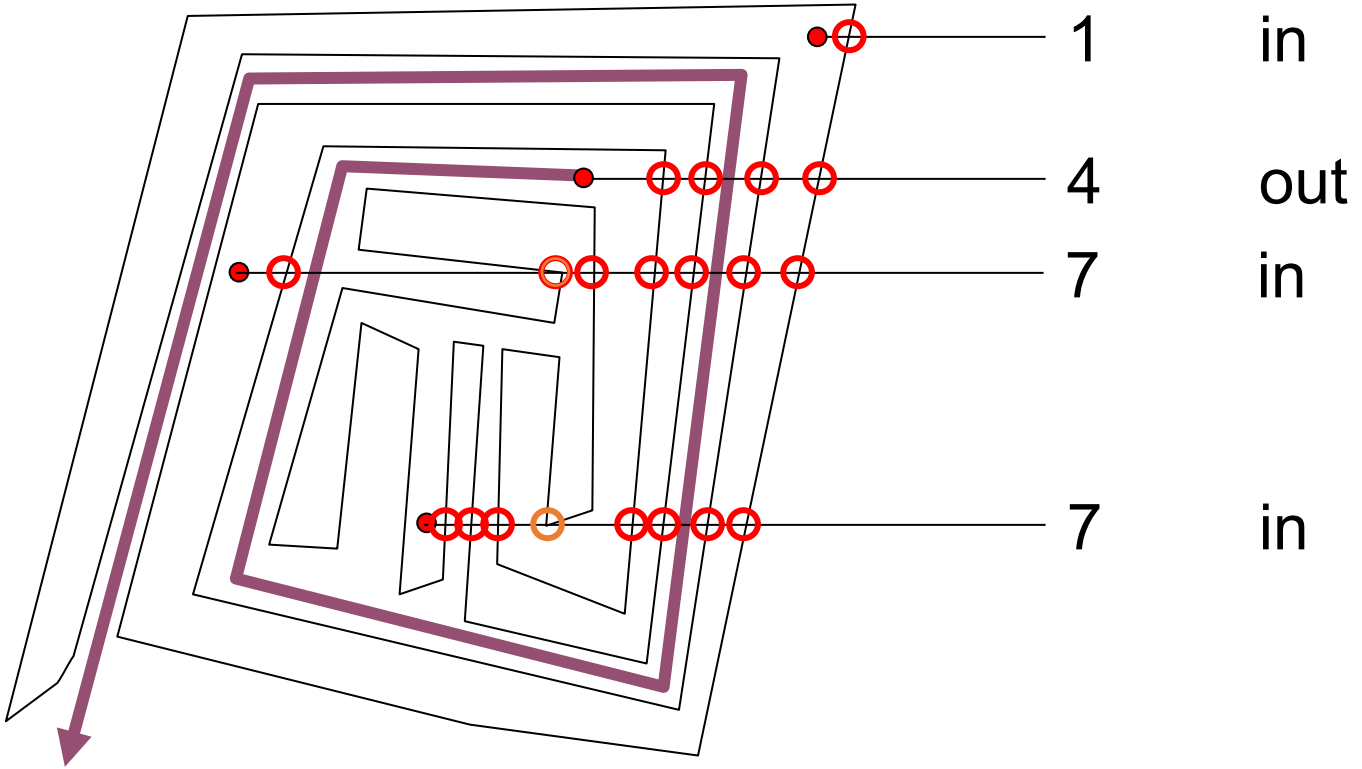
BSP-trees for the Concave Polyhedron proxy



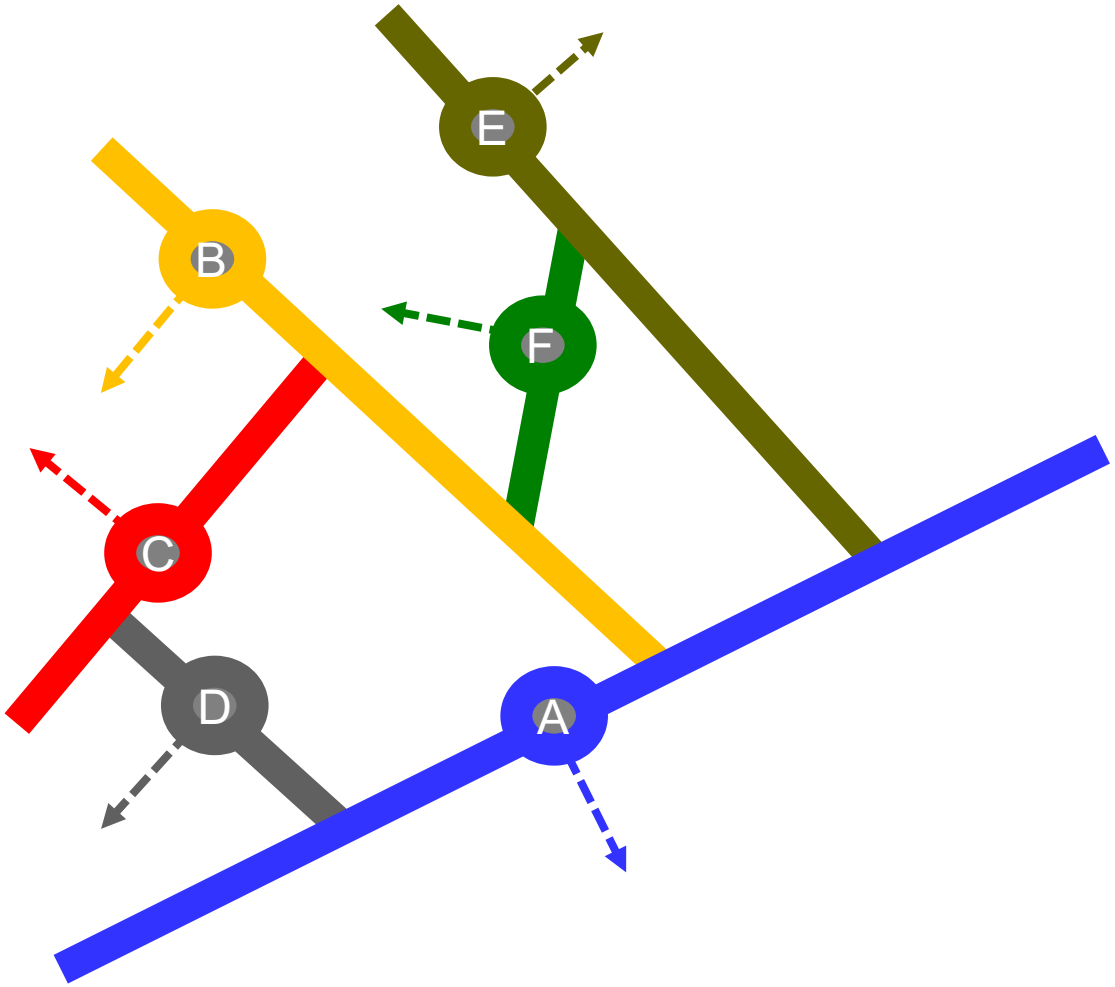
Concave Polygons: Odd-Even Test

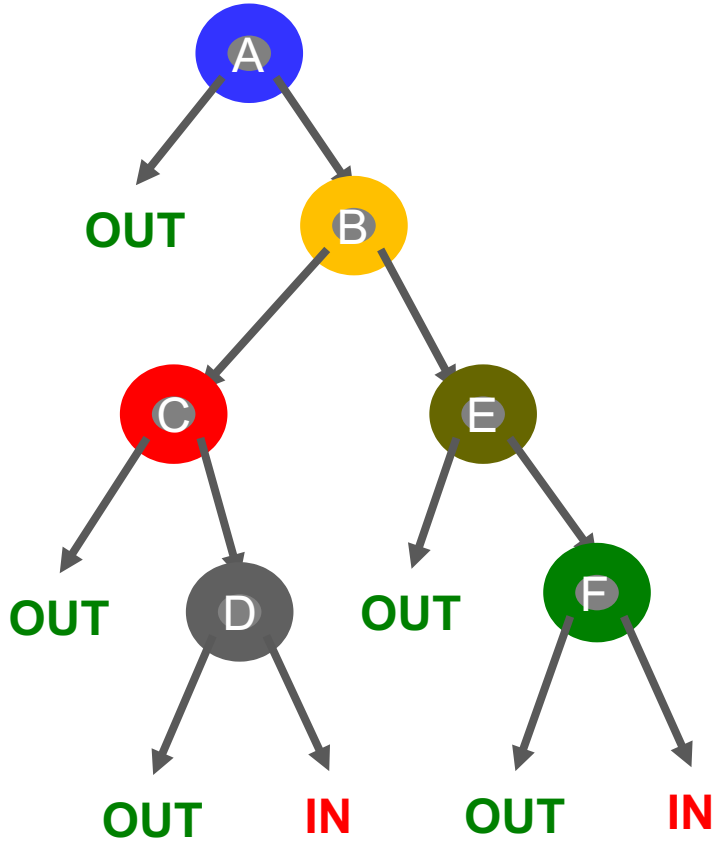
- For each scan line
  - Find all scan line/polygon intersections
  - Sort them left to right
  - Fill the interior spans between intersections
- Parity rule: inside after an odd number of crossings





BSP-trees for Inside-Outside Test

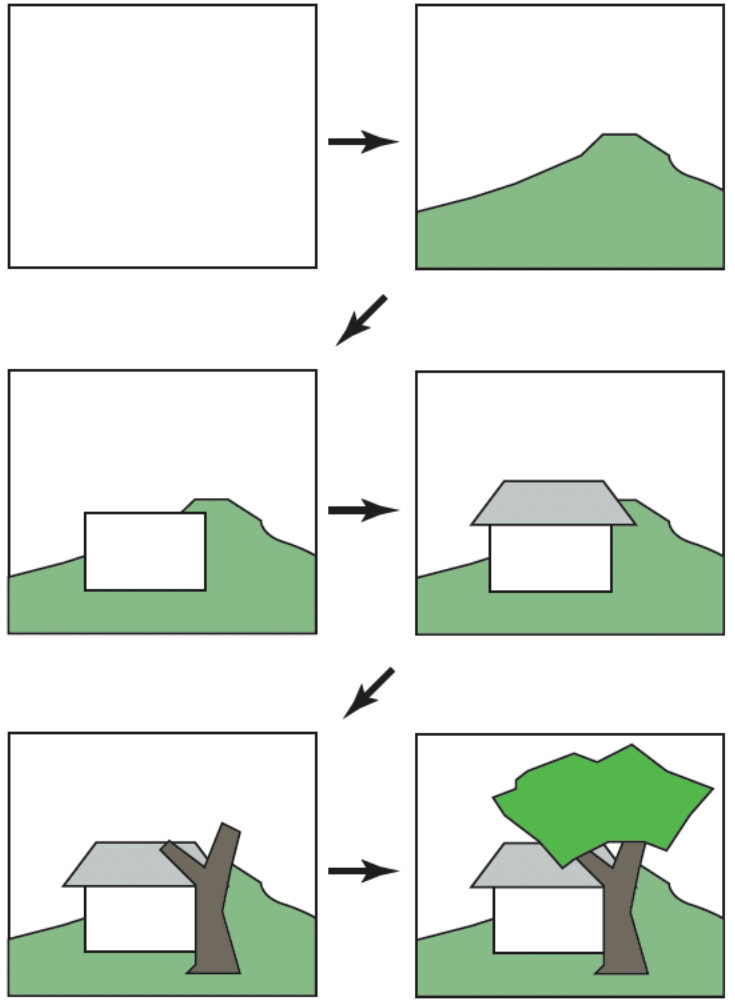


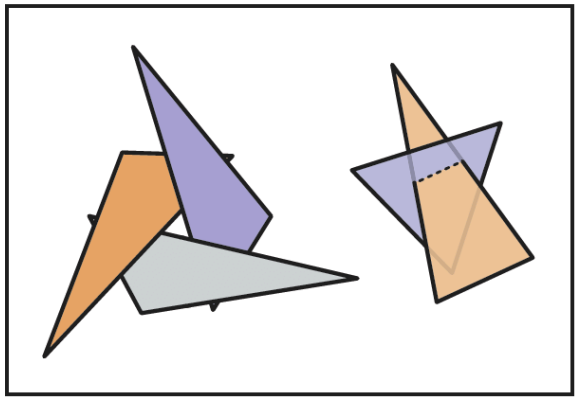


painter's algorithm

- sort objects back to front relative to viewpoint
- for each object do
  - draw object on screen

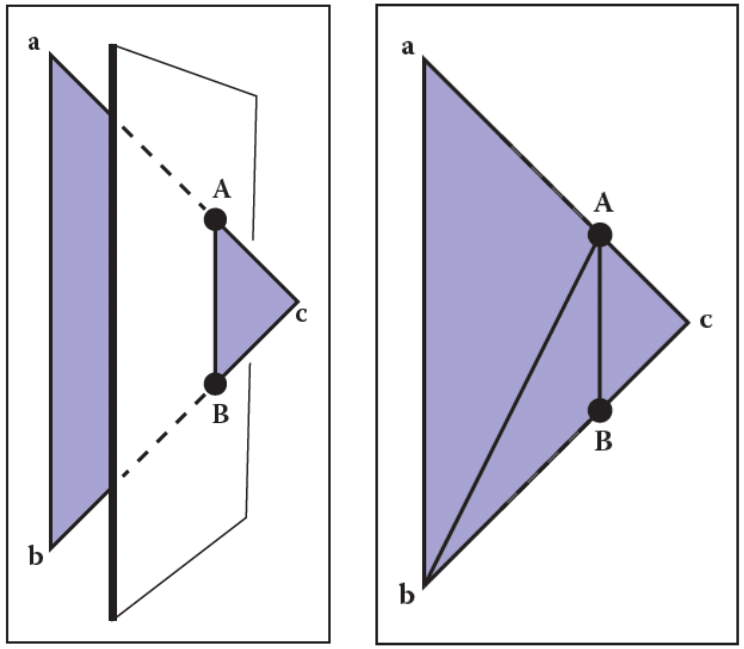
- How to implement it?
  - BSP





Painter's Algorithm with BSP Trees

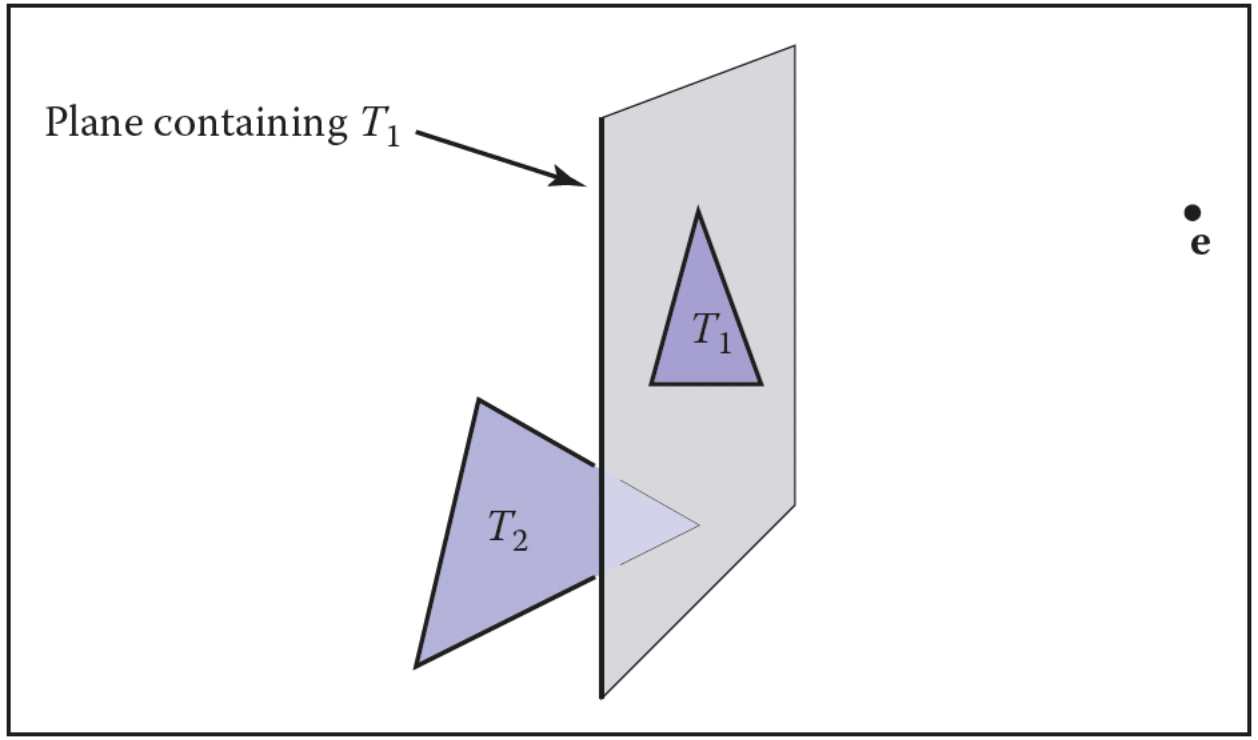
- The BSP tree algorithm works on any scene composed of polygons where no polygon crosses the plane defined by any other polygon.
- This restriction is then relaxed by a preprocessing step: cutting triangles.



The basic idea

Assume that that T2 is on the f1(p) < 0

if $(f_1(\mathbf{e}) < 0)$ **then**
    draw T_1
    draw T_2
else
    draw T_2
    draw T_1



Generalize to many objects

```
function draw(bsptree tree, point e)
    if (tree.empty) then
        return
    if (f_tree.root(e) < 0) then
        draw(tree.plus, e)
        rasterize tree.triangle
        draw(tree.minus, e)
    else
        draw(tree.minus, e)
        rasterize tree.triangle
        draw(tree.plus, e)
```

BSP-tree
Binary Spatial Partitioning tree

- Another variant
  - a binary tree (like the kD-tree)
  - but, each node is split by an **arbitrary** plane (or a **line**, in 2D)
    - plane is stored at node, as (nx, ny, nz, k)
  - planes can be optimized for a given scene
    - e.g. to go for a 50%-50% object split at each node

- Another use: to test (Generic) Polyhedron proxy:
  - note: with planes defined in its **object space**
  - each leaf: inside or outside (no need to store them: left-child = in, right-child = out)
  - tree precomputed for a given object

Summary of accelerating geometric queries: choose the right structure for the job

- Primitive vs. spatial partitioning:
  - Primitive partitioning: partition sets of objects
    - Bounded number of BVH nodes, simpler to update if primitives in scene change position
  - Spatial partitioning: partition space
    - Traverse space in order (first intersection is closest intersection), may intersect primitive multiple times
- Adaptive structures (BVH, K-D tree)
  - More costly to construct (must be able to amortize construction over many geometric queries)
  - Better intersection performance under non-uniform distribution of primitives
- Non-adaptive accelerations structures (uniform grids)
  - Simple, cheap to construct
  - Good intersection performance if scene primitives are uniformly distributed
- Many, many combinations thereof

Spatial Indexing Structures

- Regular Grid
  - the most parallelizable (to update / construct / use)
  - constant time access (best!)
  - quadratic / cubic space (2D, 3D)
- kD-tree, Oct-tree, Quad-tree
  - Compact, simple
  - non constant accessing time (still logarithmic on average)
- BSP-tree
  - optimized splits! best performance when accessed
  - optimized splits! more complex construction / update
  - ideal for static parts of the scene
  - (also, used for visibility, generic polyhedron inside/outside test, etc)
- BVH
  - simplest construction
  - non necessarily very efficient to access
    - may need to traverse multiple children
    - if you do not have a scene-graph you need to create one
  - ideal for dynamic parts of the scene

Real-Time and Interactive Ray Tracing

- Interactive ray tracing via space subdivision
  http://www.cs.utah.edu/~reinhard/egwr/

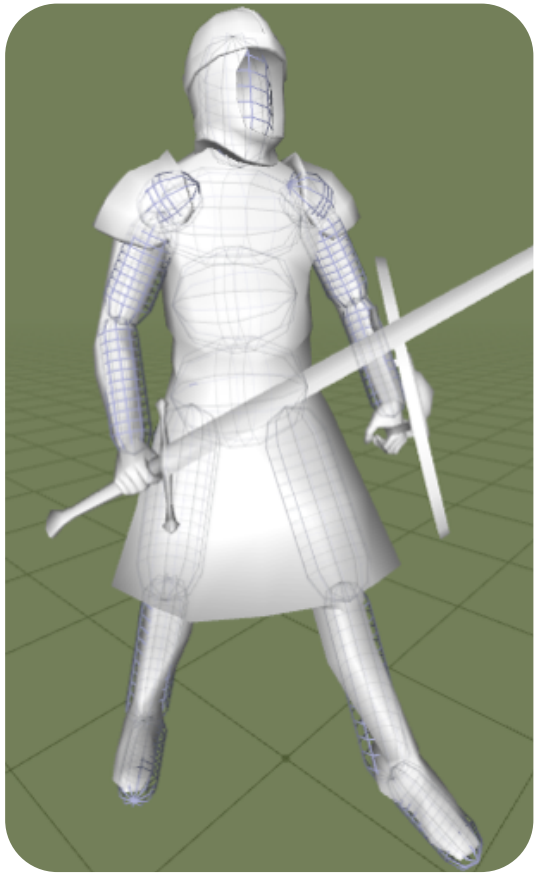
- State of the art in interactive ray tracing
  http://www.cs.utah.edu/~shirley/irt/

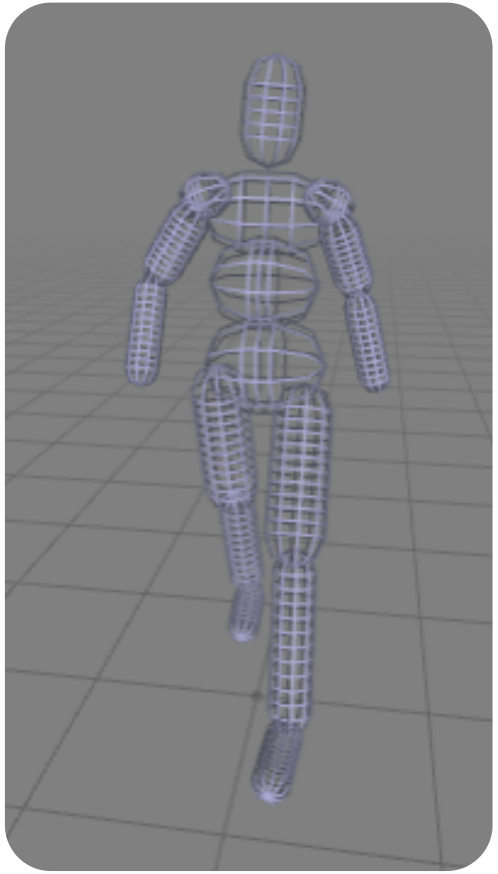
Collision Detection

- It is easy to do, the challenge is to do it efficiently
- An observation:
  - most pair of objects do not intersect each other in a scene, **collisions are rare**
  - optimizing the intersections directly is important but not sufficient, we need to optimize the detection of non intersecting pairs (“**early rejects**”)

Geometric Proxies

- Idea: use a geometric proxy to approximate the objects in the scene



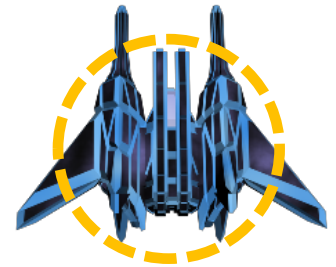


Geometric Proxy

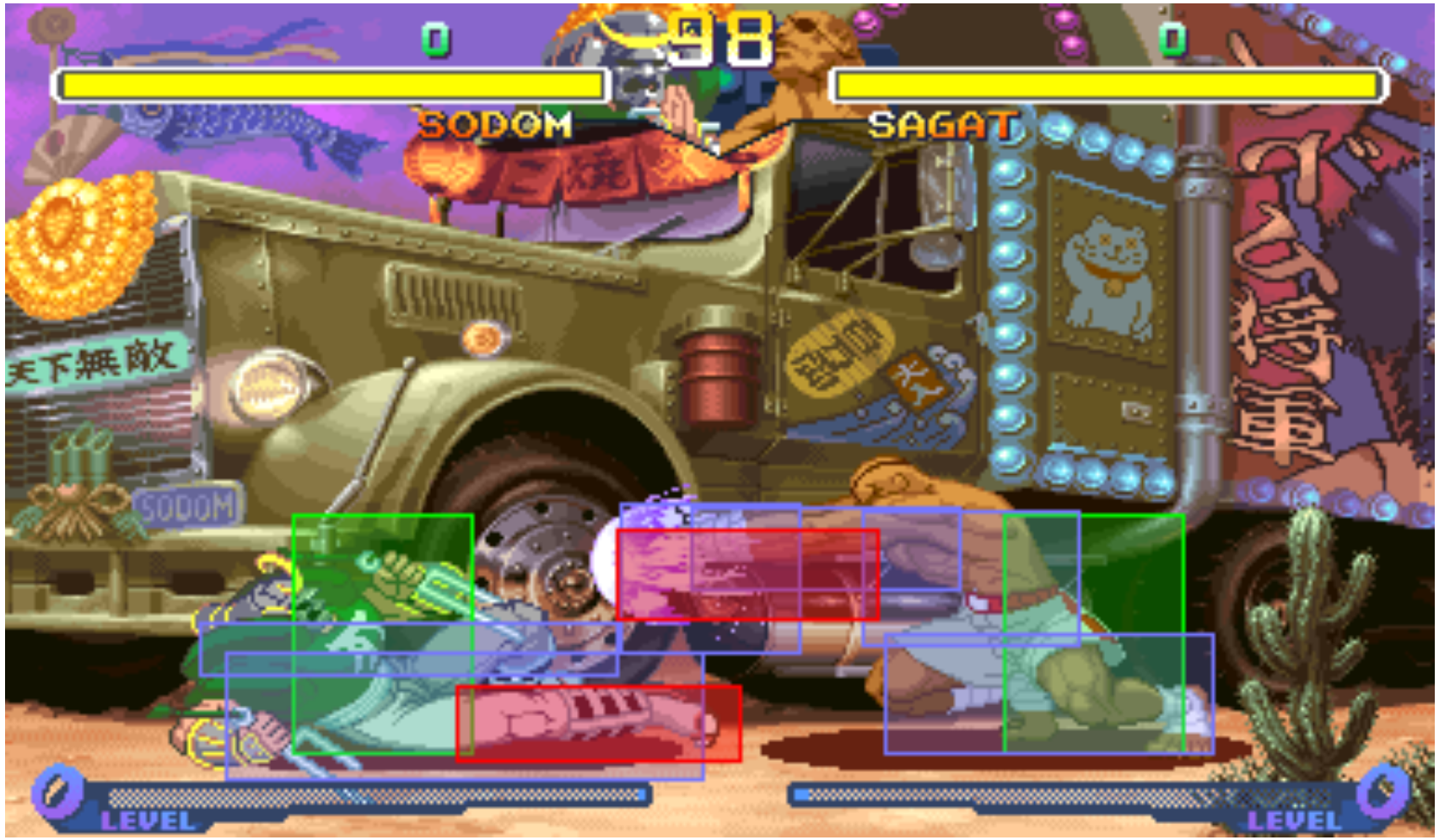
- Extremely coarse approximation

- Used as a:
  - Bounding Volume
    - the entire object must be contained inside
    - exact result, you need to do more work if you detect a collision

  - Collision Object (or “hit-box”)
    - approximation of the object
    - no need to do anything else if an approximation is ok for your use case



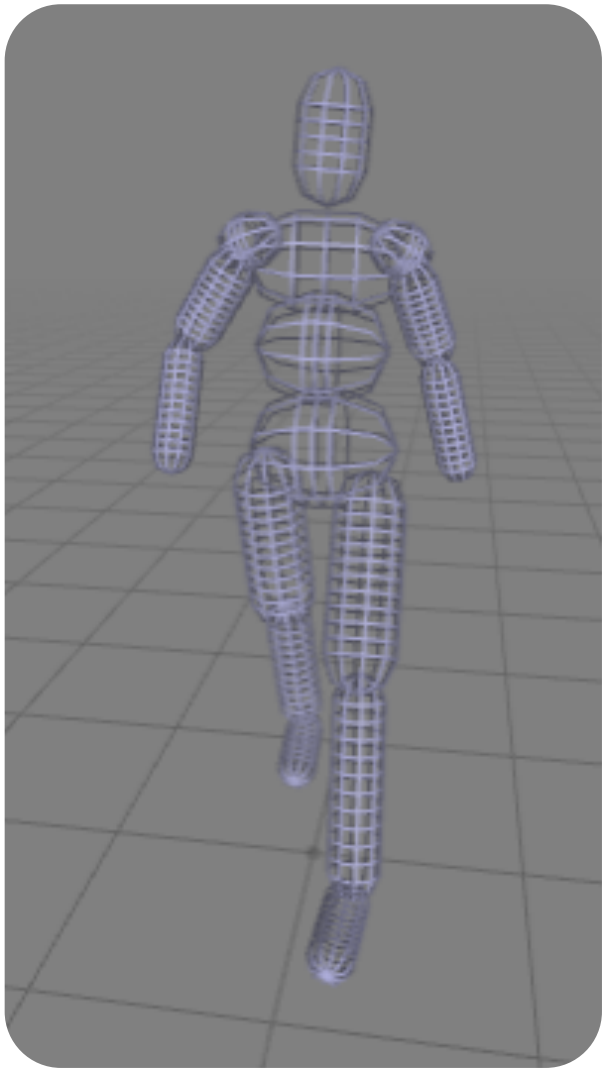
Example: Fighting Games



Street Fighter Alpha, CAPCOM 1995

Geometric Proxy is Extremely Common

- Physic engine
  - collision detection
  - collision response
- Rendering
  - view frustum culling
  - occlusion culling
- AI
  - visibility test
- GUI
  - picking

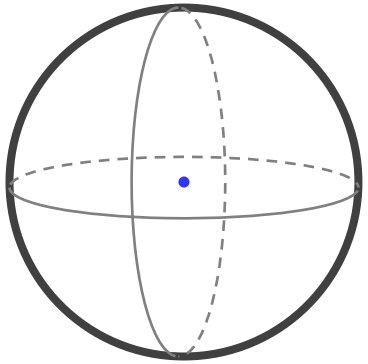


Properties of Geometric Proxies

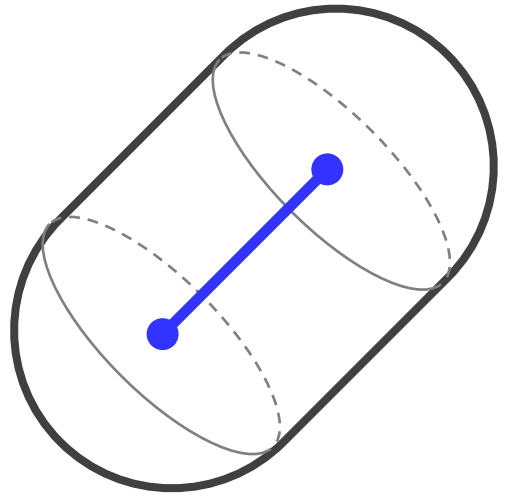
1. How expensive are they to compute/update?
2. How much space do you need?
3. Are they invariant to the transformations applied on the object?
4. How good is the approximation?
5. How expensive are the collision queries with the other objects in the scene?

Geometry Proxies: Sphere

- Easy to compute and update
- Compact (center, radius)
- Very efficient collision tests
- Can only be transformed rigidly
- The quality of the approximation is low



Geometry Proxies: Capsule



- Sphere == points with dist from a point < radius
- Capsule == points with dist from a segment < radius

- Stored with:
  - a segment (two end-points)
  - a radius (a scalar)

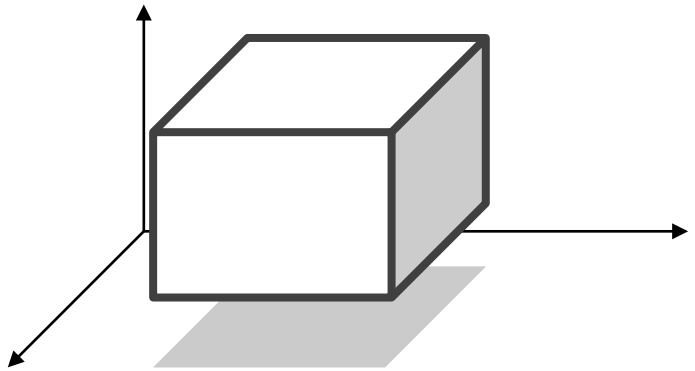
- Popular option, compact to store, easy to construct, easy to detect intersections, good approximation

Geometry Proxies: Half Space

- Trivial, but useful
  - e.g. for a flat terrain,
    or a wall
- Storage:
  - (nx, ny, nz, k)
  - a normal, a distance from the origin
- Tests are trivial

Geometry Proxies:
Axis-Aligned Bounding Box (AABB)

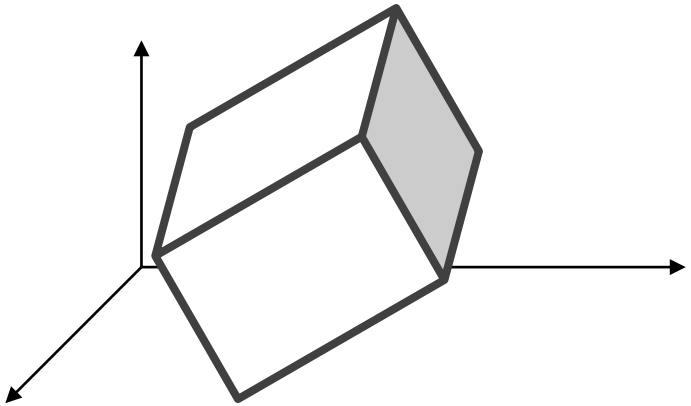
- Easy to update
- Compact (three intervals)
- Trivial to test



- It can only be translated or scaled, rotations are not supported

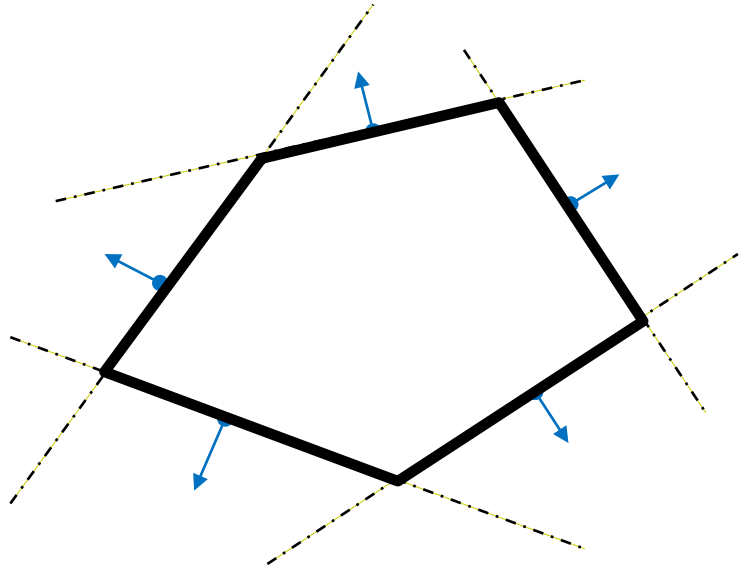
Geometry Proxies: Box

- Similar to AABB, but not axis-aligned
- More expensive to compute and store
  - You need intervals and a rotation
- Still not a great approximation, but it is invariant to rotations and it is fast to compute and use



Geometry Proxies (in 2D): Convex Polygon

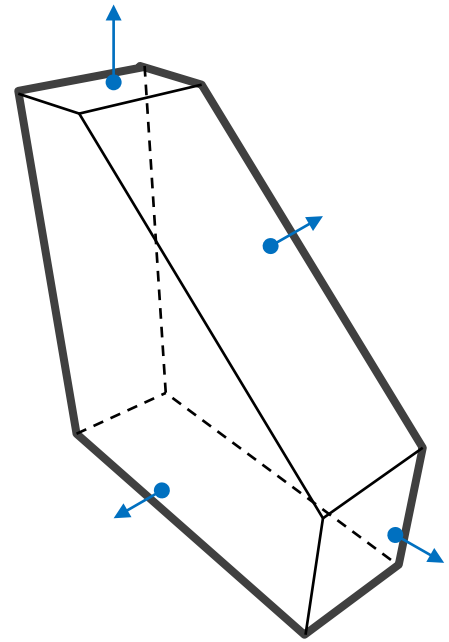
- Intersection of half-planes
  - each delimited by a line
- Stored as:
  - a collection of (oriented) lines
- Test:
  - a point is inside iff it is in each half-plane
- Good approximation
- Moderate complexity



Geometry Proxies (in 3D): Convex Polyhedron

- Intersection of half-spaces

- Similar as previous, but in 3D
  - Stored as a collection of planes
  - Each plane is a normal + distance from origin
  - Test: inside proxy iff inside each half-space



Geometry Proxies (in 3D): (General) Polyhedron

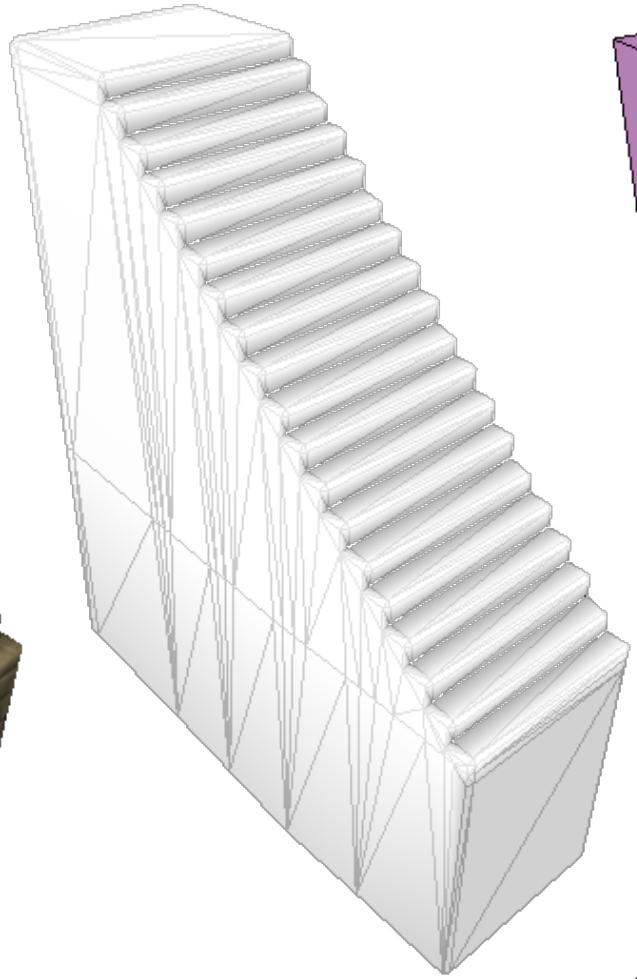
- Luxury Hit-Boxes :)
  - The most accurate approximations
  - The most expensive tests / storage
- Specific algorithms to test for collisions
  - requiring some preprocessing
  - and data structures (BSP-trees)
- Creation (as meshes):
  - sometimes, with automatic simplification
  - often, hand made (low poly modelling)

3D Meshes as Hit-Boxes

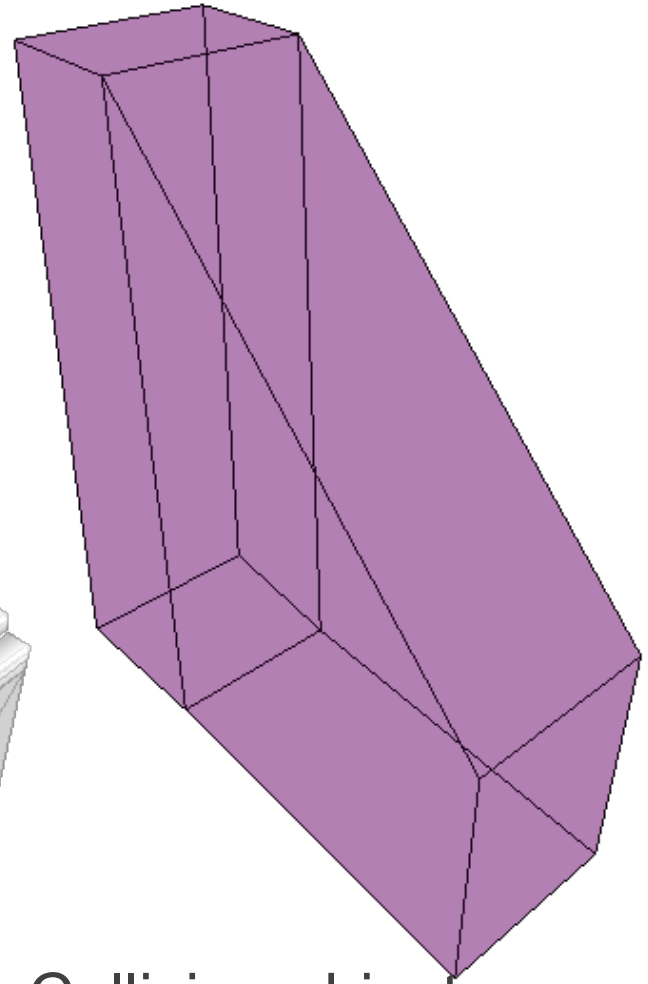
- These are often NOT the meshes that you use for rendering
  - much lower resolution (~ $O(10^2)$)
  - no attributes (no uv-mapping, no col, etc)
  - closed, water-tight (inside != outside)
  - often convex only
  - can be polygonal (as long as the faces are flat)

3D Meshes as Hit-Boxes

mesh for rendering
(~600 tri faces)



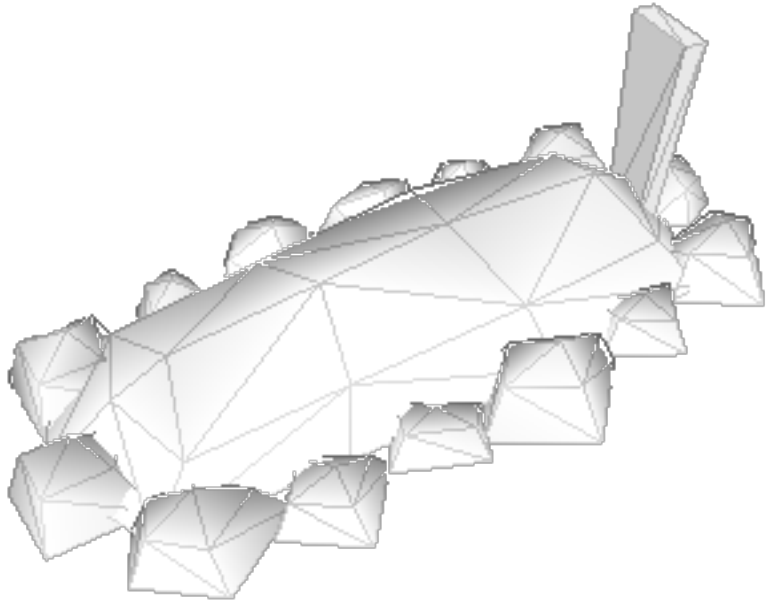
(in wireframe)



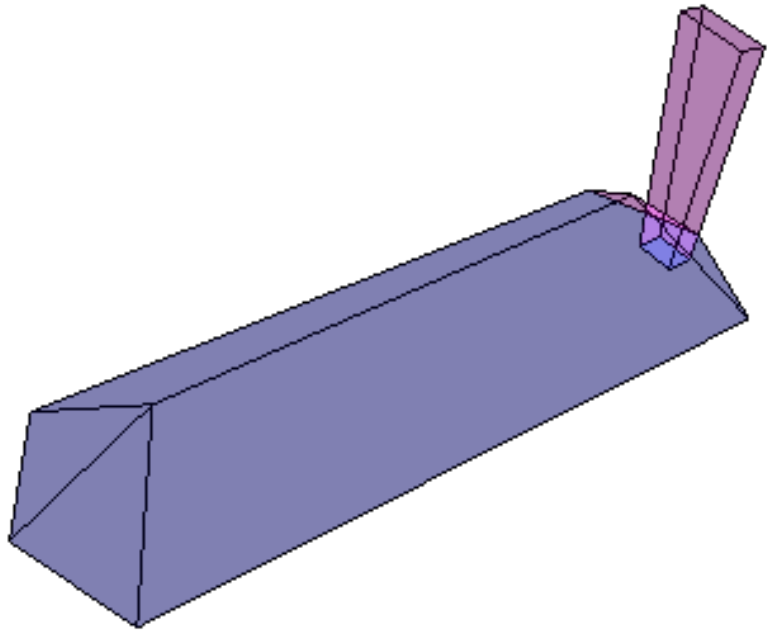
Collision object:
10 (polygonal) faces

3D Meshes as Hit-Boxes

mesh for rendering
(~300 tri faces)



(in wireframe)



Collision object:
12 (polygonal) faces

Geometry Proxies: Composite Hit-Boxes

- Union of Hit-Boxes
  - inside *iff* inside of *any* sub Hit-Box
- Flexible
  - union of **convex** Hit-Boxes ==> **concave** Hit-Box
  - shape partially defined by a sphere,
    partially by a box ==> better approximation
- Creation: typically by hand
  - (remember: hit-boxes are usually **assets**)

How To Choose The Proxy?

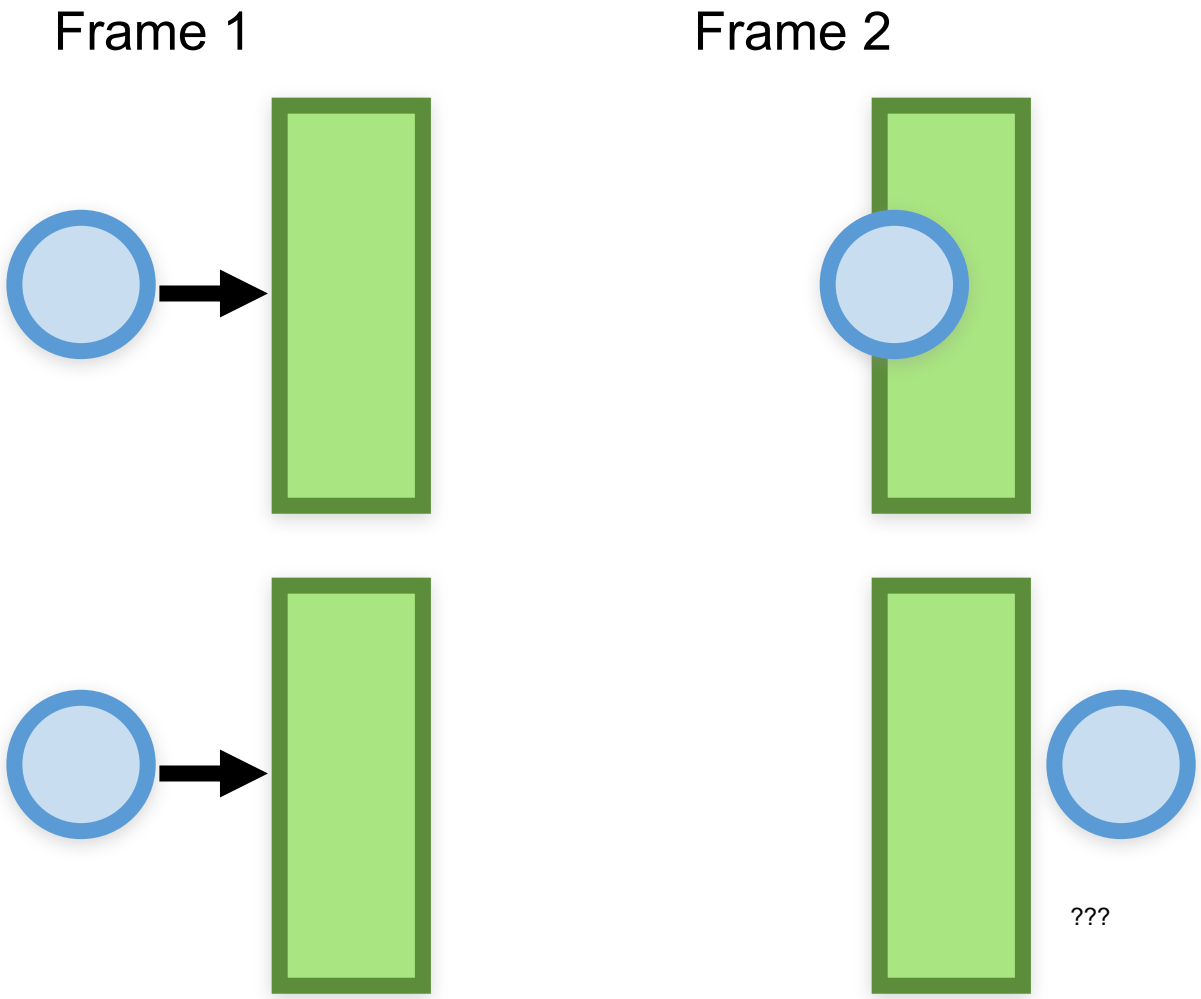
- Application dependent
- Note: # of intersection tests to be implemented quadratic wrt # of types supported

VS	Type A	Type B	Type C	Point	Ray
Type A	*algorithm*	*algorithm*	*algorithm*	*algorithm*	*algorithm*
Type B		*algorithm*	*algorithm*	*algorithm*	*algorithm*
Type C			*algorithm*	*algorithm*	*algorithm*

useful,
e.g.
for visibility

Collision Detection Strategies

- **Static** Collision detection
  - (“a posteriori”, “discrete”)
  - approximated
  - simple + quick
- **Dynamic** Collision detection
  - (“a priori”, “continuous”)
  - accurate
  - demanding



Existing Implementations

- Intel Embree - BVH Tree - https://embree.github.io
- Nori - BVH - https://github.com/wjakob/nori
- Approximate knn - https://www.cs.umd.edu/~mount/ANN/
- Intersections - http://www.realtimerendering.com/intersections.html

References

Foundations of Multidimensional and Metric Data Structures
Hanan Samet

http://www.realtimerendering.com/books.html

http://www.realtimerendering.com/intersections.html

Polygon Mesh Processing
Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, Bruno Levy

Fundamentals of Computer Graphics, Fourth Edition
4th Edition **by** Steve Marschner, Peter Shirley
Chapter 12