Computer Graphics -Geometry Queries

Junjie Cao @ DLUT

Spring 2017

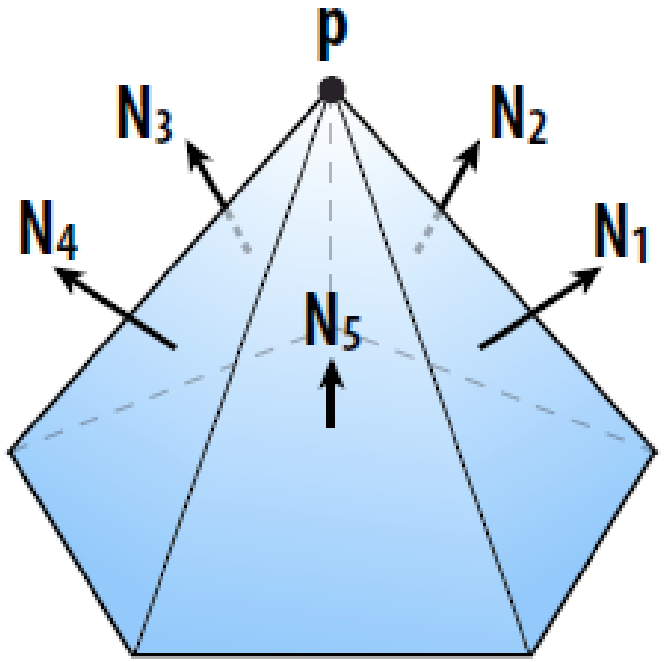
http://jjcao.github.io/ComputerGraphics/

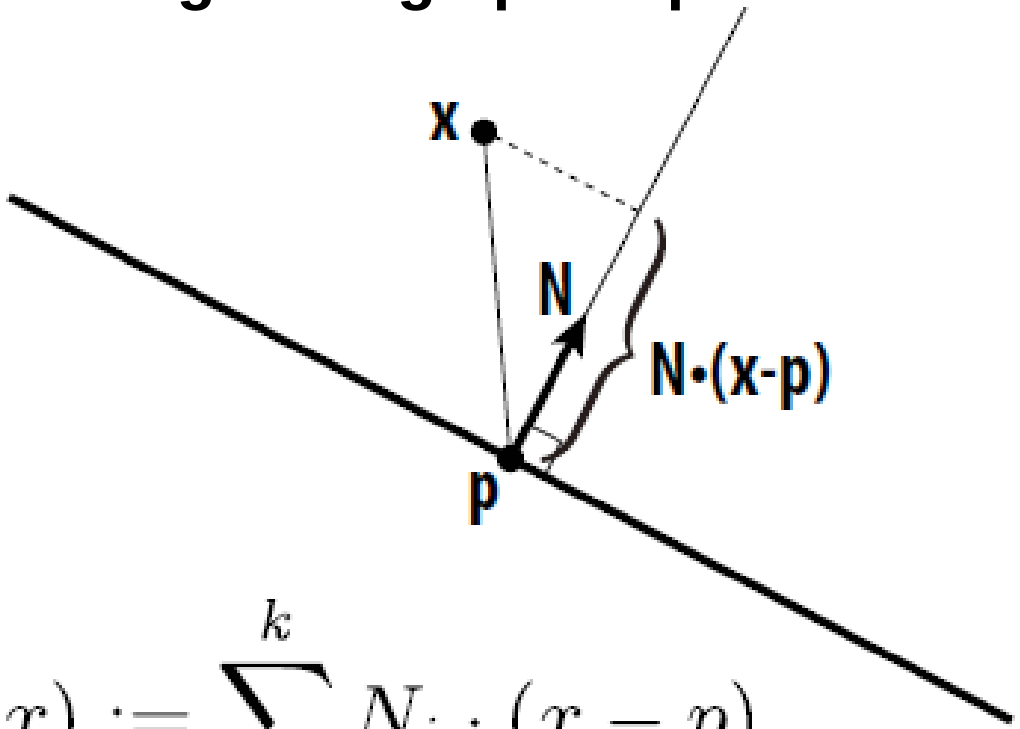
Simplification via Quadric Error Metric

- One popular scheme: iteratively collapse edges
- Which edges? Assign score with *quadric error metric**
  - approximate distance to surface as sum of distance to aggregated triangles
  - iteratively collapse edge with smallest score
  - greedy algorithm... great results!

Quadric Error Metric

- Approximate distance to a collection of triangles
- Distance is sum of point-to-plane distances
  - Q: Distance to plane with normal N passing through point p?
  - A: d(x) = N•x - N•p = N•(x-p)
- Sum of distances:





$$d(x) := \sum_{i=1}^{k} N_i \cdot (x - p)$$

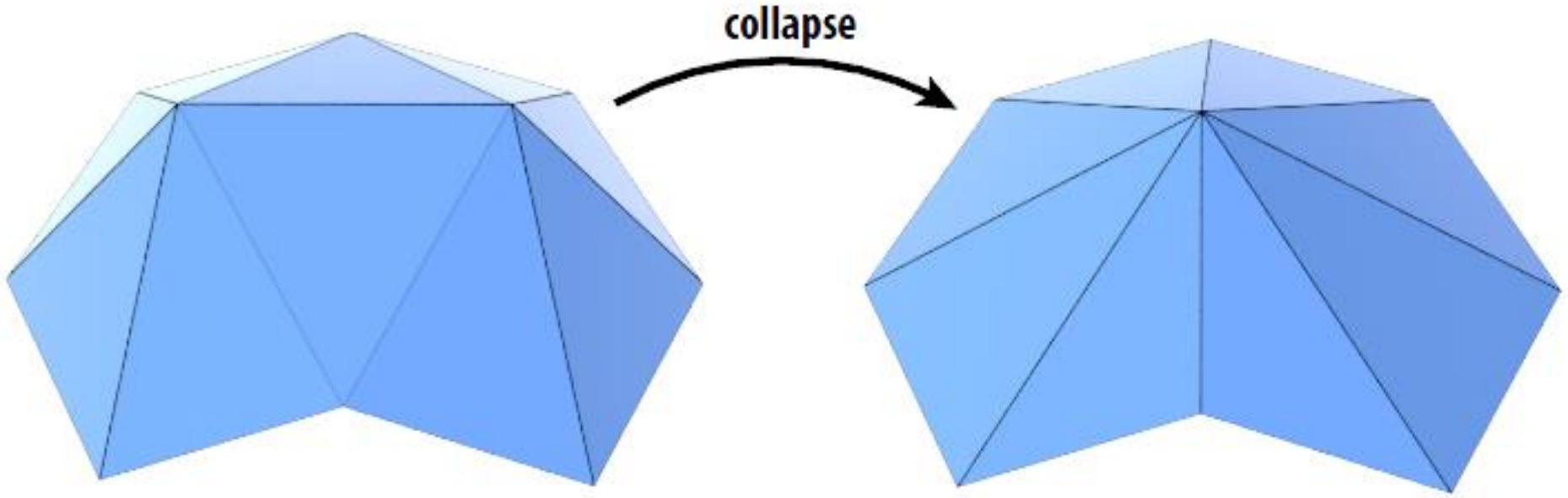
Quadric Error - Homogeneous Coordinates

- **Suppose in coordinates we have**
  - **a query point (x,y,z)**
  - **a normal (a,b,c)**
  - **an offset d := -(x,y,z) • (a,b,c)**
- **Then in homogeneous coordinates, let**
  - **u := (x,y,z,1)**
  - **v := (a,b,c,d)**
- ***Signed* distance to plane is then just u•v = ax+by+cz+d**
- ***Squared* distance is (u'v)2 = u'(vv')u =: u'Qu**
- **Key idea: *matrix Q encodes distance to plane***
- **Q is symmetric, contains 10 unique coefficients (small storage)**

$$Q = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

Quadric Error of Edge Collapse

- How much does it cost to collapse an edge?
- Idea: compute edge midpoint, measure quadric error



- Better idea: use point that minimizes quadric error as new point!
- Q: How do we minimize quadric error?

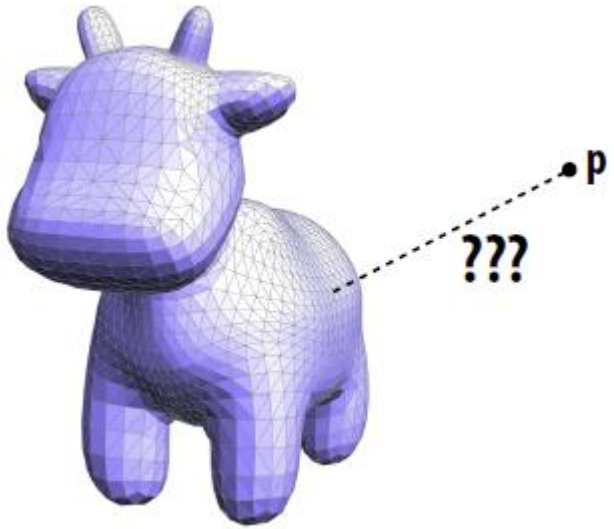
But wait: we have the original mesh.

Why not just project each new sample point

onto the closest point of the original mesh?

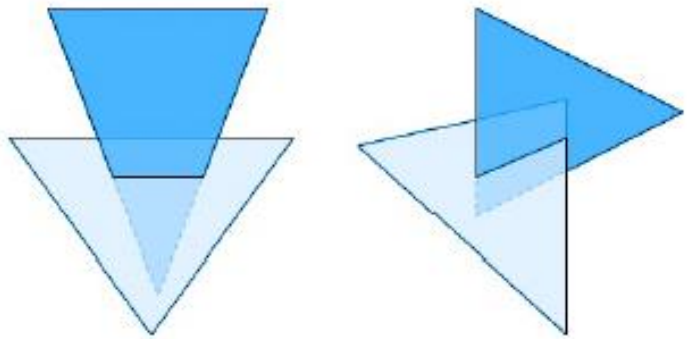
Geometric Queries

- Q: Given a point, in space (e.g., a new sample point), how do we find the closest point on a given surface?
- Q: Does implicit/explicit representation make this easier?
- Q: Does our halfedge data structure help?
- Q: What's the cost of the naïve algorithm?
- Q: How do we find the distance to a single triangle anyway?
- So many questions!



Many types of geometric queries

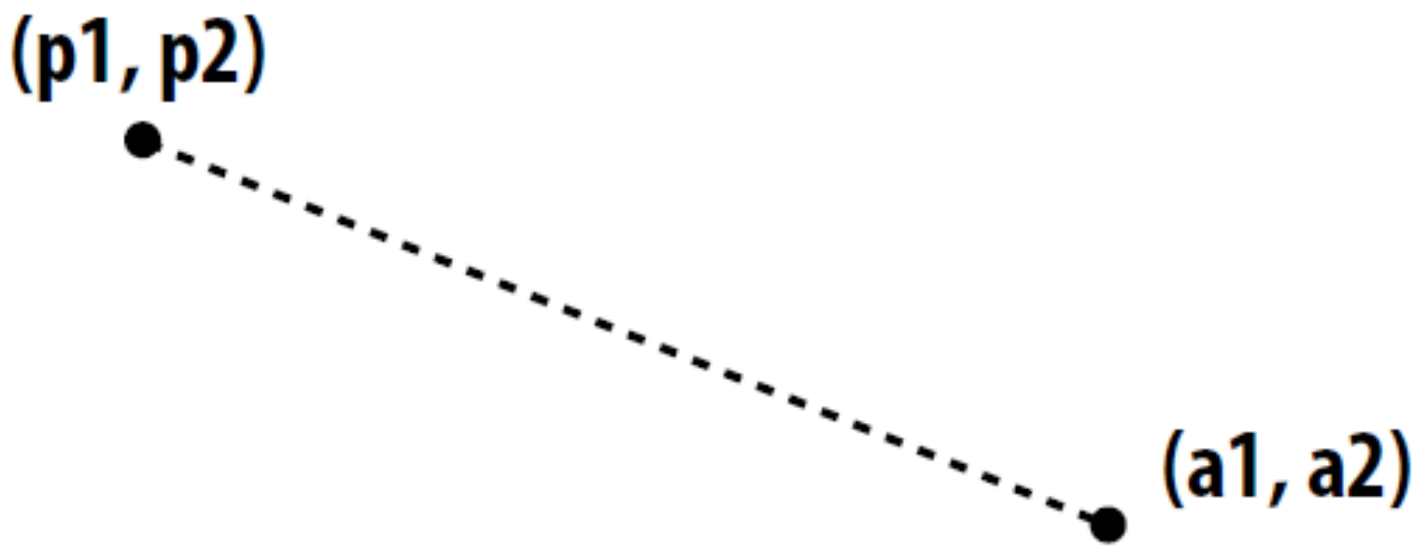
- Already identified need for “closest point” query
- Plenty of other things we might like to know:
  - Do two triangles intersect?
  - Are we inside or outside an object?
  - Does one object contain another?
  - ...



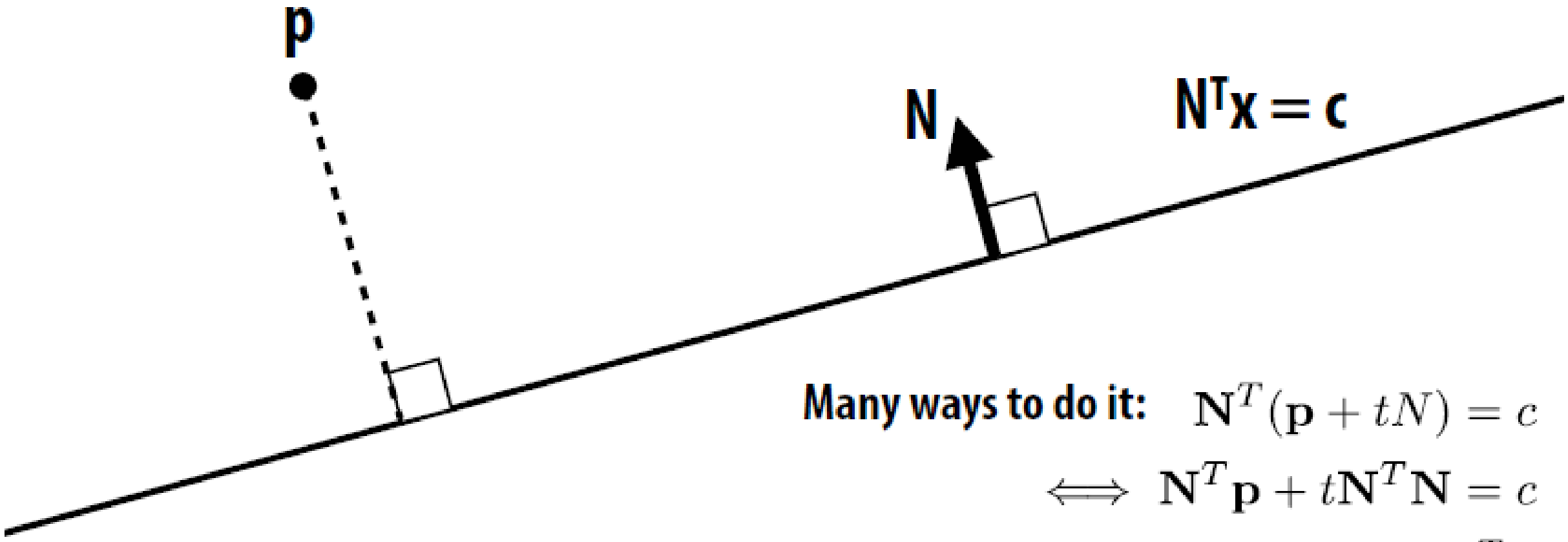
- Data structures we’ve seen so far not really designed for this...
- Need some new ideas!
- Today: come up with simple (read: slow) algorithms.
- Then: intelligent ways to accelerate geometric queries.

Warm up: closest point on point

- **Goal is to find the point on a mesh closest to a given point.**
- ***Much* simpler question: given a query point (p1,p2), how do we find the closest point on the point (a1,a2)?**



Slightly harder: closest point on line



Many ways to do it:

$$\mathbf{N}^T(\mathbf{p} + tN) = c$$

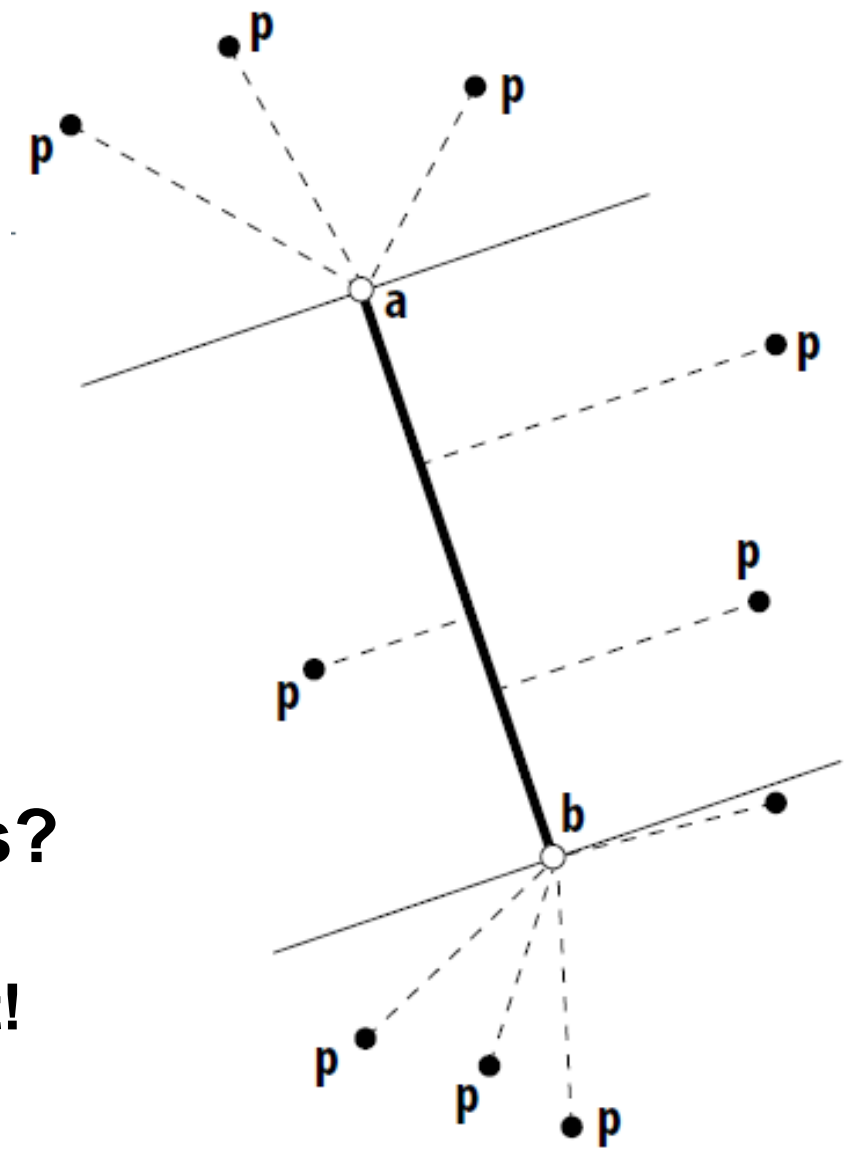
$$\iff \mathbf{N}^T\mathbf{p} + t\mathbf{N}^T\mathbf{N} = c$$

$$\iff t = c - \mathbf{N}^T\mathbf{p}$$

$$\Rightarrow \mathbf{p} + t\mathbf{N} = \boxed{\mathbf{p} + (c - \mathbf{N}^T\mathbf{p})\mathbf{N}}$$

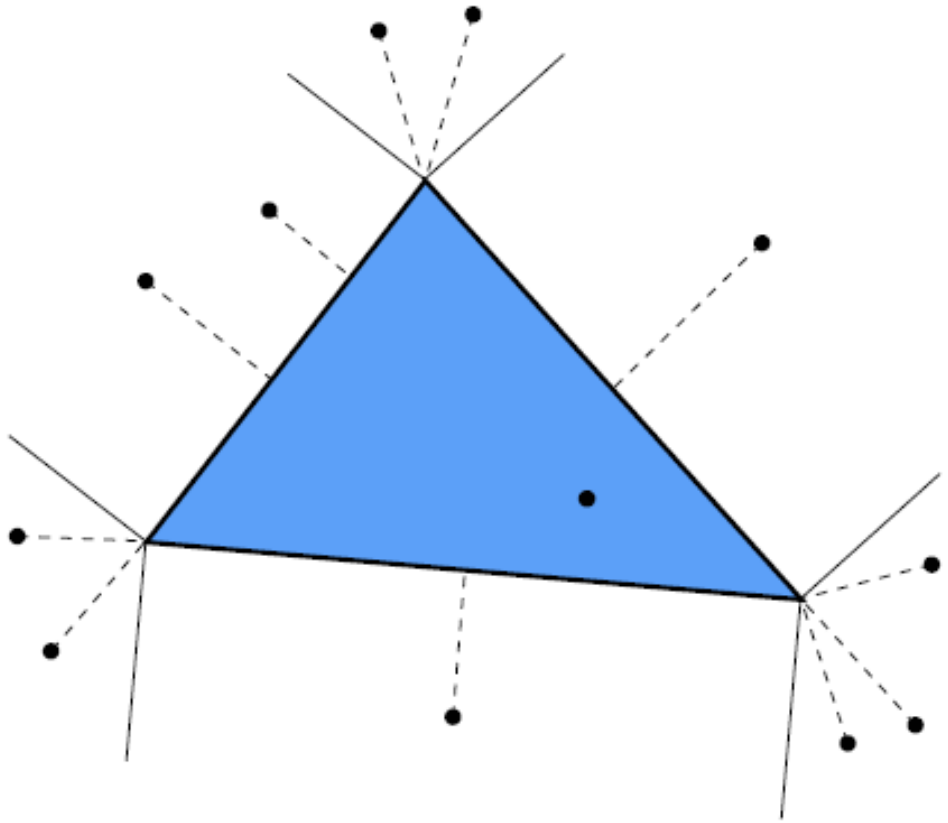
Harder: closest point on line segment

- Two cases: endpoint or interior
- Already have basic components:
  - point-to-point
  - point-to-line
- Algorithm?
  - find closest point on line
  - check if it's between endpoints
  - if not, take closest endpoint
- How do we know if it's between endpoints?
  - write closest point on line as a+t(b-a)
  - if t is between 0 and 1, it's inside the segment!



Even harder: closest point on triangle in 2D

- **What are all the possibilities for the closest point?**
- **Almost just minimum distance to three segments:**



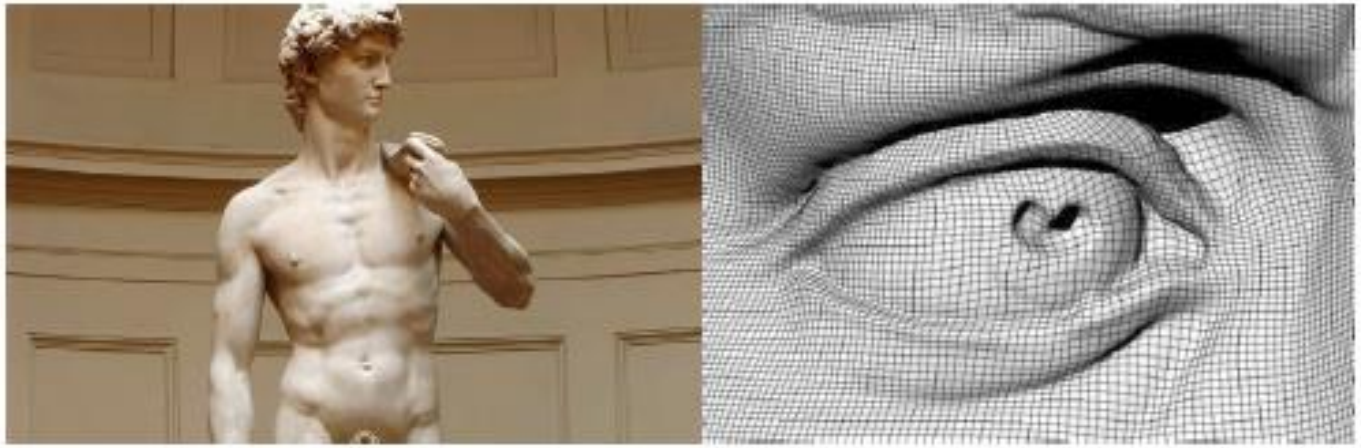
Question: what about a point inside the triangle?

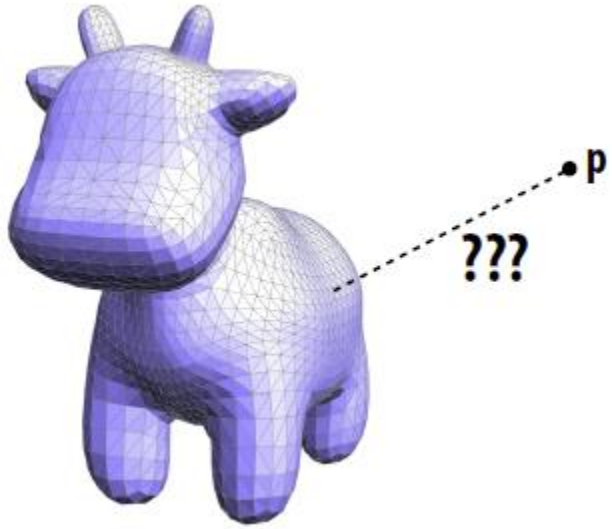
Closest point on triangle in 3D

- Not so different from 2D case
- Algorithm?
  - project onto plane of triangle
  - use half-plane tests to classify point
  - if inside the triangle, we’re done!
  - otherwise, find closest point on associated vertex or edge
- By the way, how do we find closest point on plane?
- Same expression as closest point on a line!
  - E.g., p + (c - NTp) N

Closest point on triangle *mesh* in 3D?

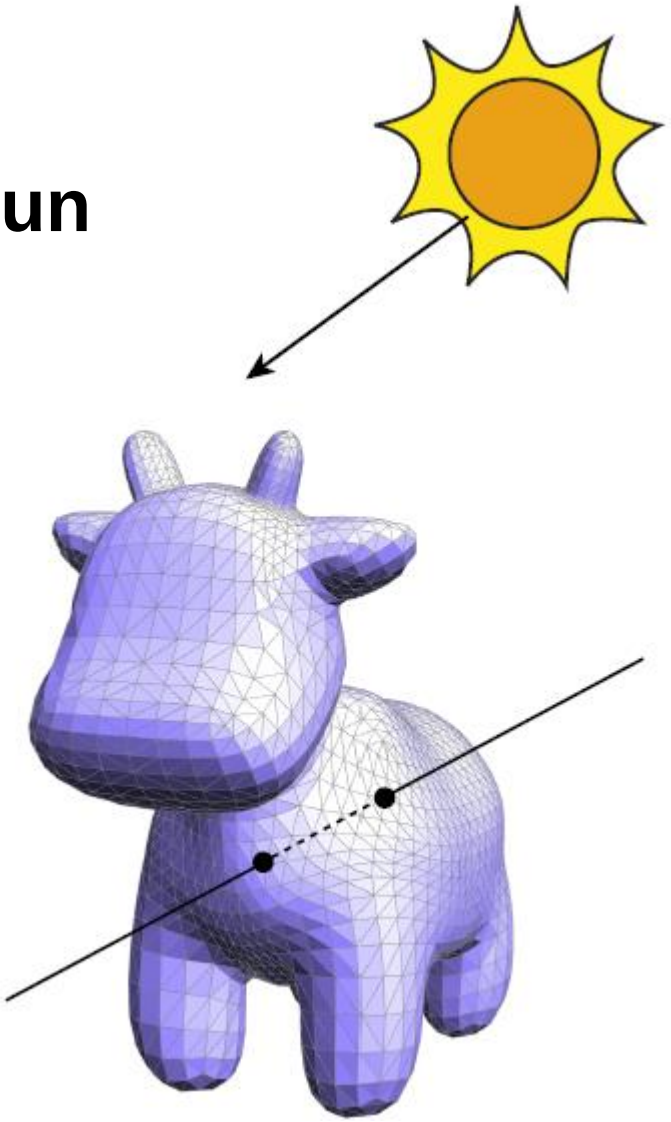
- **Conceptually easy:**
  - **loop over all triangles**
  - **compute closest point to current triangle**
  - **keep globally closest point**
- **Q: What's the cost? Does halfedge help?**
- **What if we have *billions* of faces?**





Different query: ray-mesh intersection

- A “ray” is an oriented line starting at a point
- Think about a ray of light traveling from the sun
- Want to know where a ray pierces a surface
- Why?
  - GEOMETRY: inside-outside test
  - RENDERING: visibility, ray tracing
  - SIMULATION: collision detection
- Might pierce surface in many places!



Ray equation

- Can express ray as

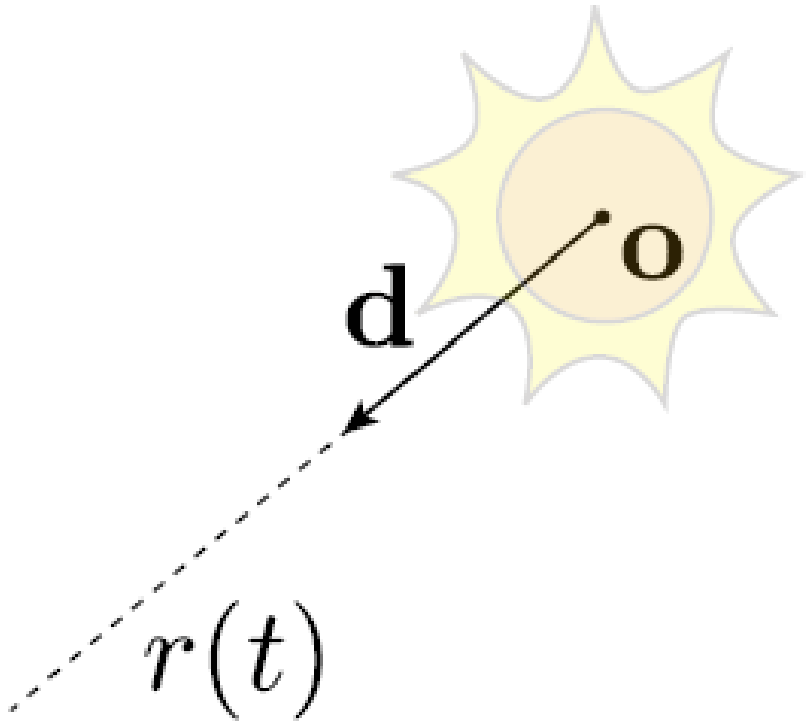
origin

unit direction

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

point along ray

"time"



Intersecting a ray with an implicit surface

- Recall implicit surfaces: all points x such that f(x) = 0
- Q: How do we find points where a ray pierces this surface?
- Well, we know all points along the ray: r(t) = o + td
- Idea: replace “x” with “r” in 1st equation, and solve for t
- Example: unit sphere

$$f(\mathbf{x}) = |\mathbf{x}|^2 - 1$$

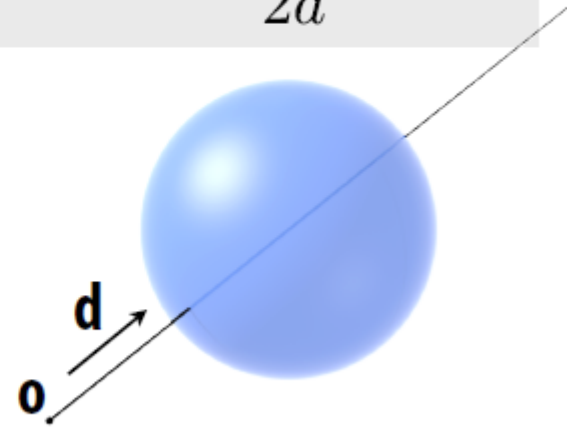
$$\Rightarrow f(\mathbf{r}(t)) = |\mathbf{o} + t\mathbf{d}|^2 - 1$$

$$\underbrace{|\mathbf{d}|^2}_{a}\, t^2 + \underbrace{2(\mathbf{o}\cdot\mathbf{d})}_{b}\, t + \underbrace{|\mathbf{o}|^2 - 1}_{c} = 0$$

$$t = \boxed{-\mathbf{o}\cdot\mathbf{d} \pm \sqrt{(\mathbf{o}\cdot\mathbf{d})^2 - |\mathbf{o}|^2 + 1}}$$

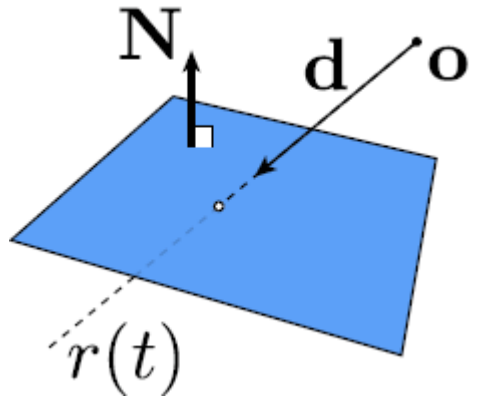
quadratic formula:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



Why two solutions?

Ray-plane intersection



- Suppose we have a plane N'x = c
  - N - unit normal
  - c - offset
- How do we find intersection with ray r(t) = o + td?
- *Key idea:* again, replace the point x with the ray equation t:

$$\mathbf{N}^\mathsf{T}\mathbf{r}(t) = c$$

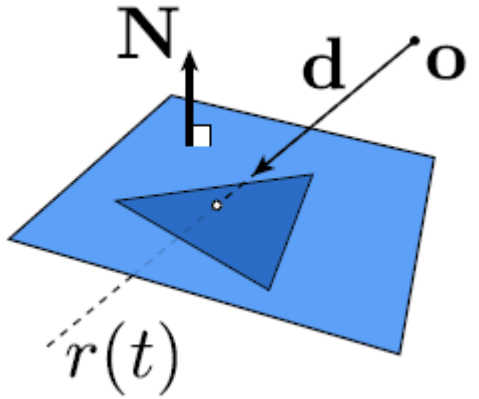
- Now solve for t:

$$\mathbf{N}^\mathsf{T}(\mathbf{o} + t\mathbf{d}) = c \qquad \Rightarrow t = \frac{c - \mathbf{N}^\mathsf{T}\mathbf{o}}{\mathbf{N}^\mathsf{T}\mathbf{d}}$$

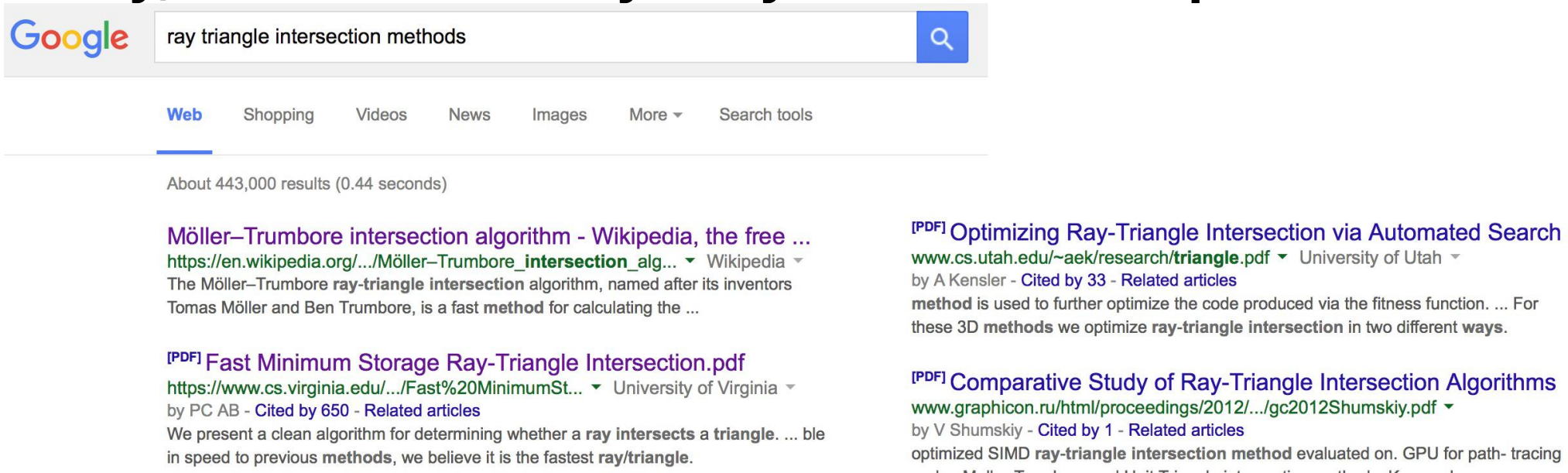
- And plug t back into ray equation:

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

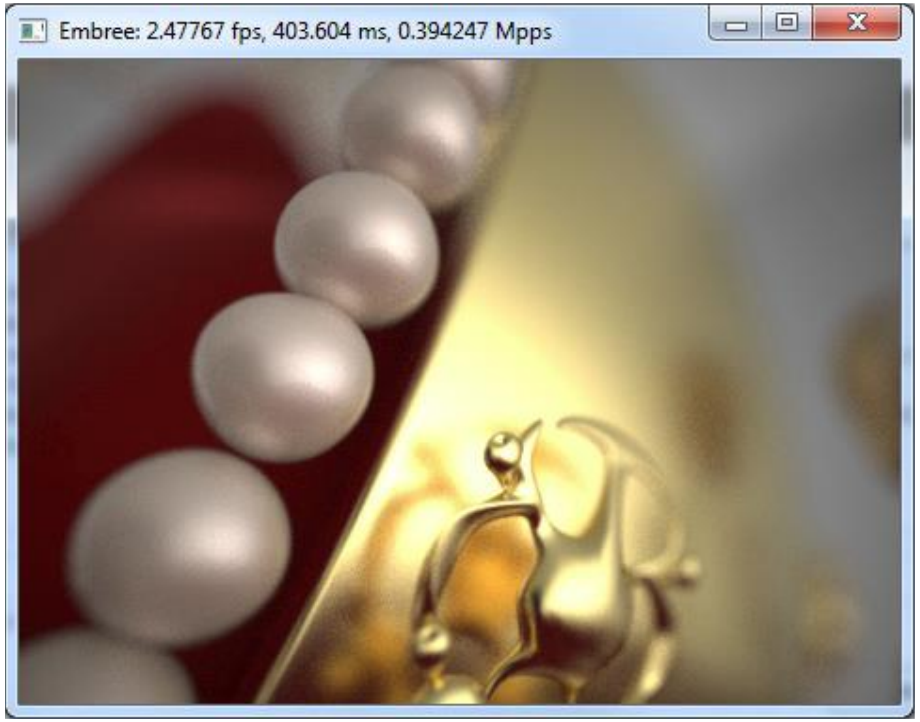
Ray-triangle intersection



- Triangle is in a plane...
- Not much more to say!
  - Compute ray-plane intersection
  - Q: What do we do now?
  - A: Why not compute barycentric coordinates of hit point?
  - If barycentric coordinates are all positive, point in triangle
- Actually, a *lot* more to say... if you care about performance!



Why care about performance?



Intel Embree

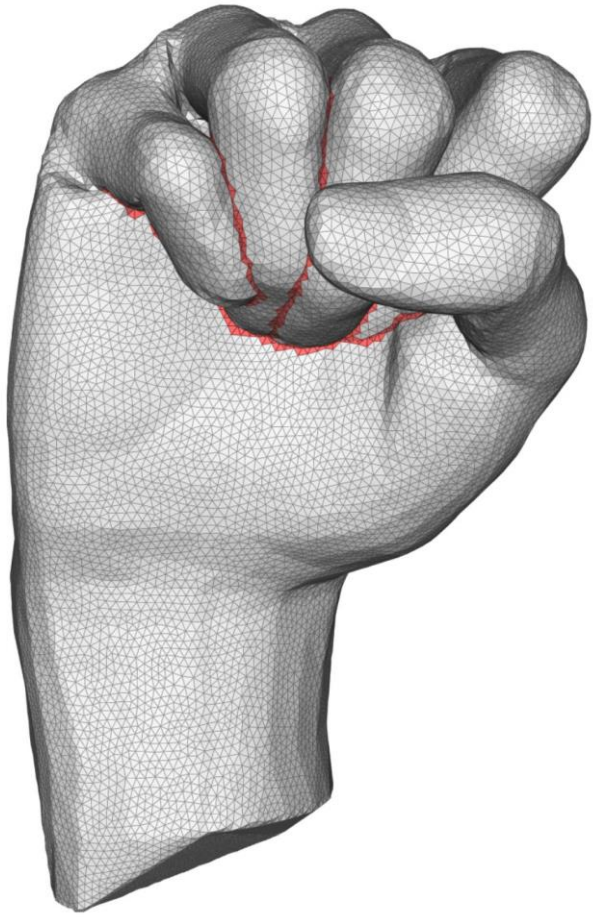
NVIDIA OptiX

Why care about performance?

“Brigade 3” real time path tracing demo

One more query: mesh-mesh intersection

- **GEOMETRY: How do we know if a mesh intersects itself?**
- **ANIMATION: How do we know if a collision occurred?**



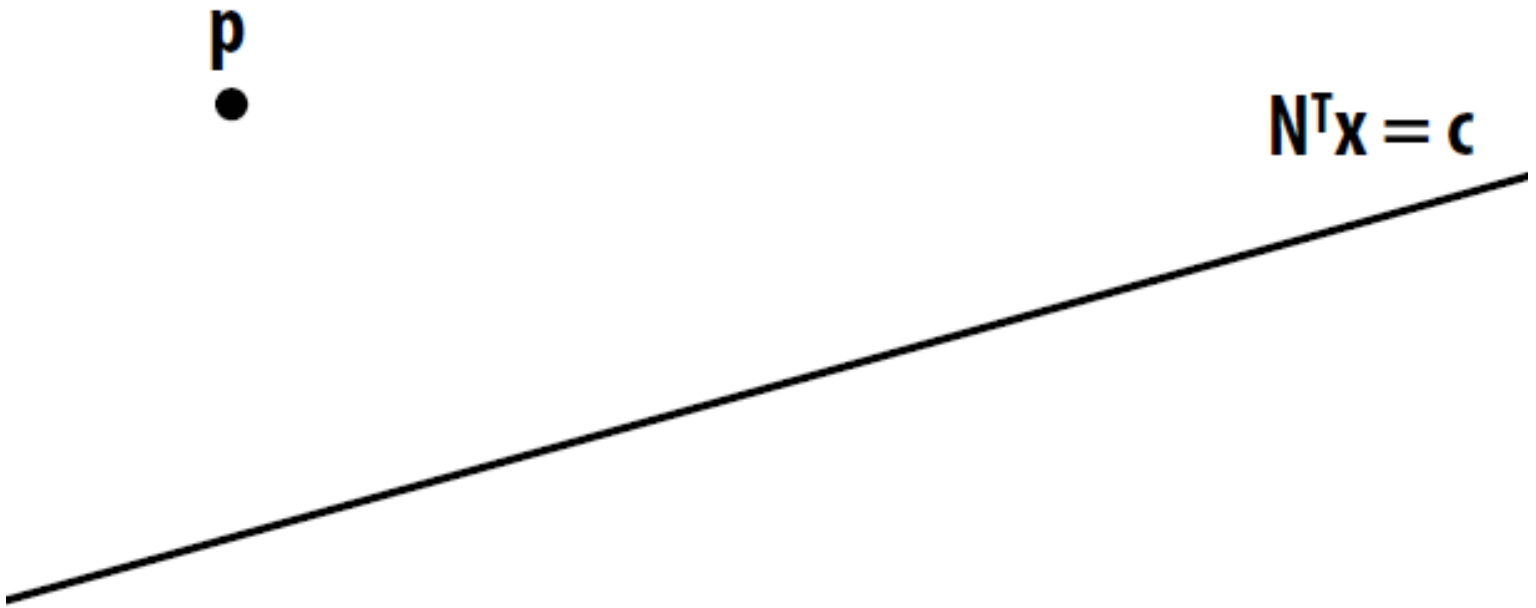
Warm up: point-point intersection

- Q: How do we know if p intersects a?
- A: ...check if they're the same point!

- Sadly, life is not always so easy.

Slightly harder: point-line intersection

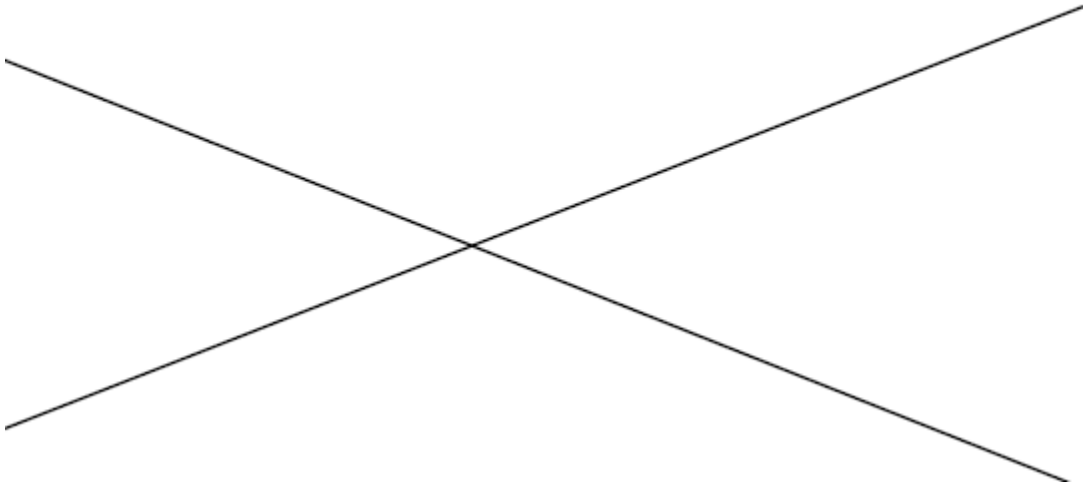
- Q: How do we know if a point intersects a given line?
- A: ...plug it into the line equation!



Finally interesting: line-line intersection

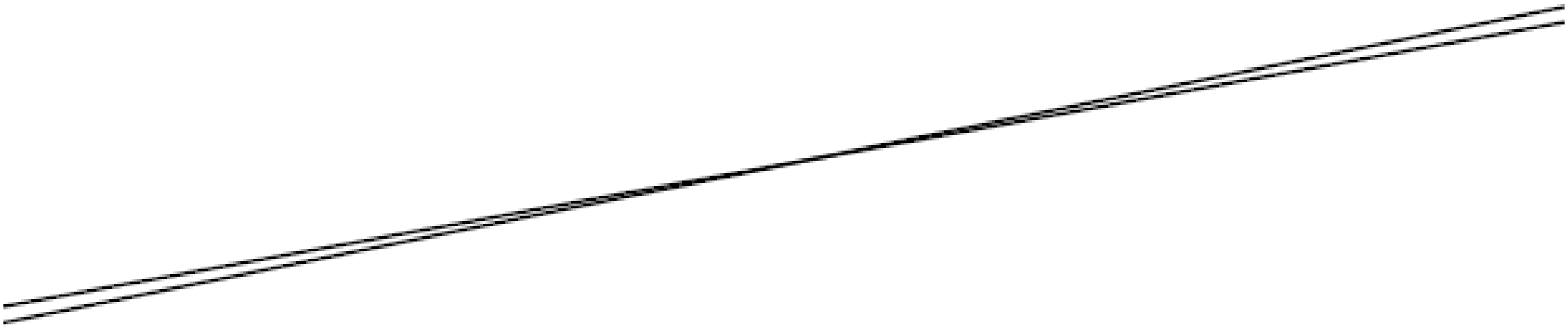
- Two lines: ax=b and cx=d
- Q: How do we find the intersection?
- A: See if there is a simultaneous solution
- Leads to linear system:

$$\begin{bmatrix} a_1 & a_2 \\ c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$$

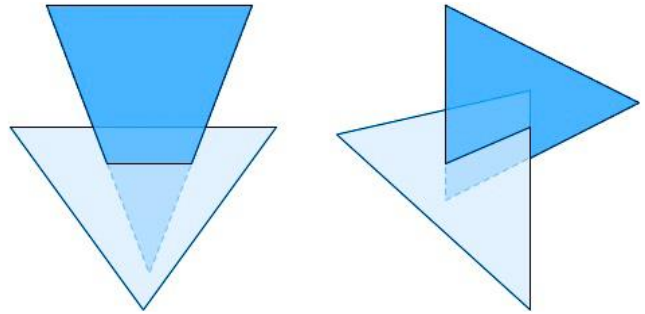


Degenerate line-line intersection?

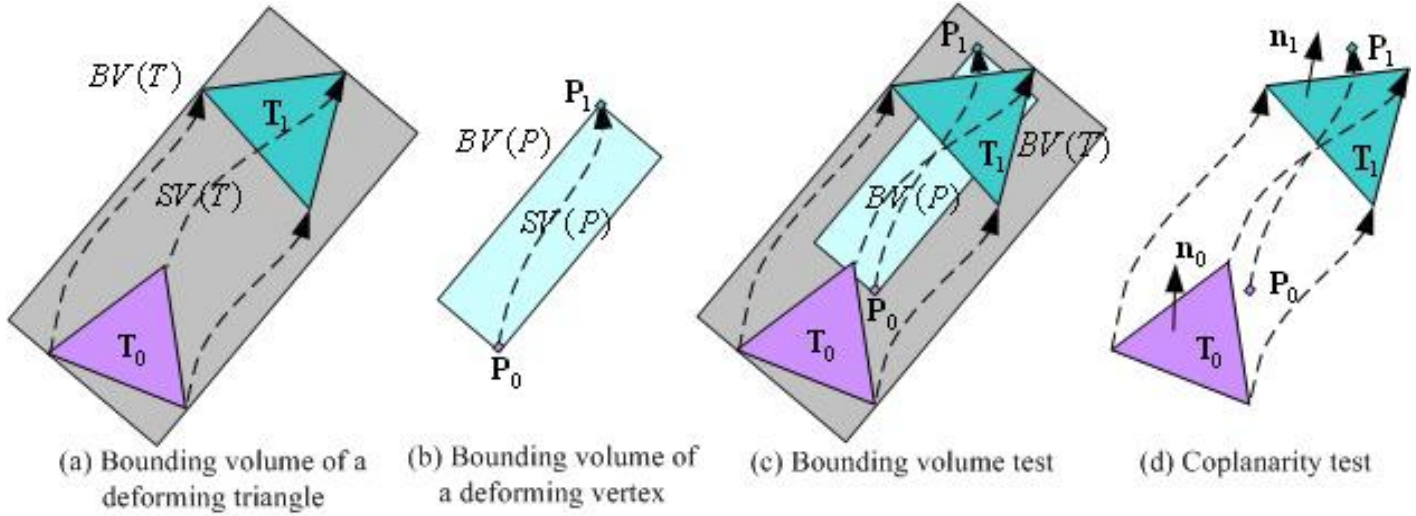
- **What if lines are almost parallel?**
- **Small change in normal can lead to big change in intersection!**
- **Instability very common, very important with geometric**
- **predicates. Demands special care (e.g., analysis of matrix).**



Triangle-Triangle Intersection?



- Lots of ways to do it
- Basic idea:
  - Q: Any ideas?
  - One way: reduce to edge-triangle intersection
  - Check if each line passes through plane
  - Then do interval test

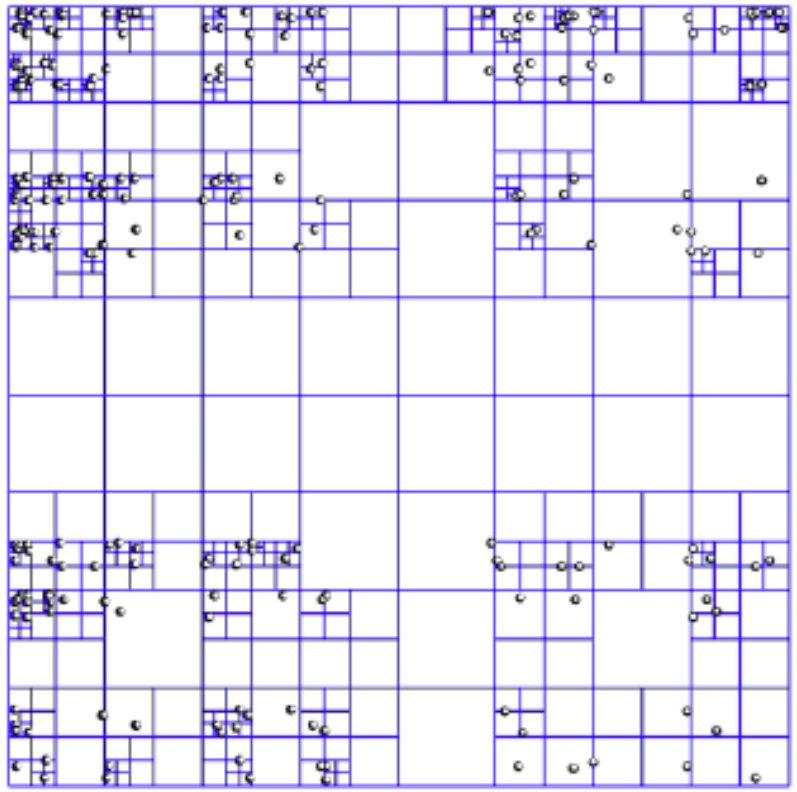


(a) Bounding volume of a deforming triangle (b) Bounding volume of a deforming vertex (c) Bounding volume test (d) Coplanarity test

- What if triangle is *moving*?
  - Important case for animation
  - Can think of triangles as *prisms* in time
  - Will say more when we talk about animation!

Up Next: Spatial Acceleration Data Strucutres

- **Testing every element is *slow!***
- **E.g., linearly scanning through a list vs. binary search**
- **Can apply this same kind of thinking to geometric queries**



Accelerating Geometric Queries

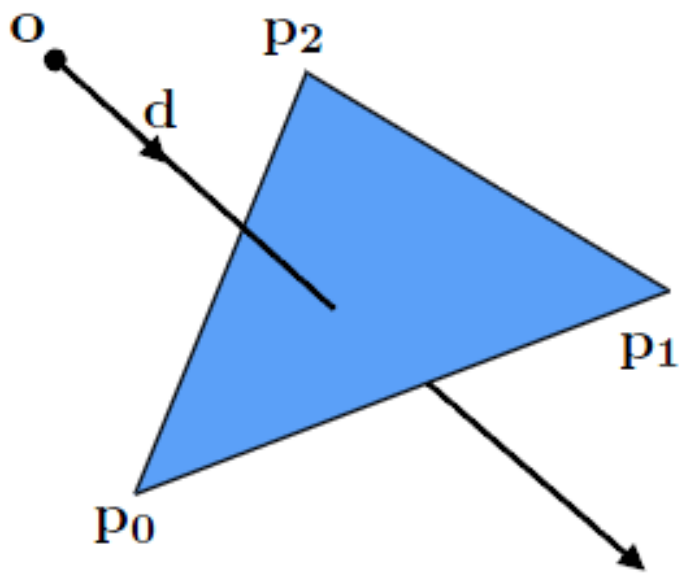
Review: ray-triangle intersection

- **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**

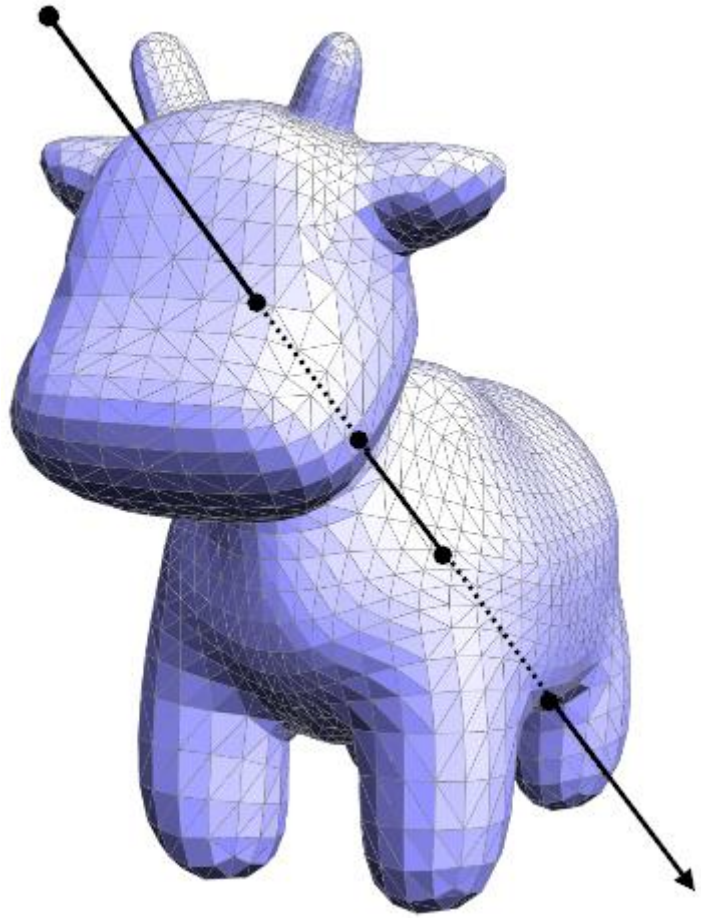
Ray-primitive queries

- **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

- 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
- “Find the first primitive the ray hits”

```
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 a new integer *k*, find the element in S that is closest to *k:***

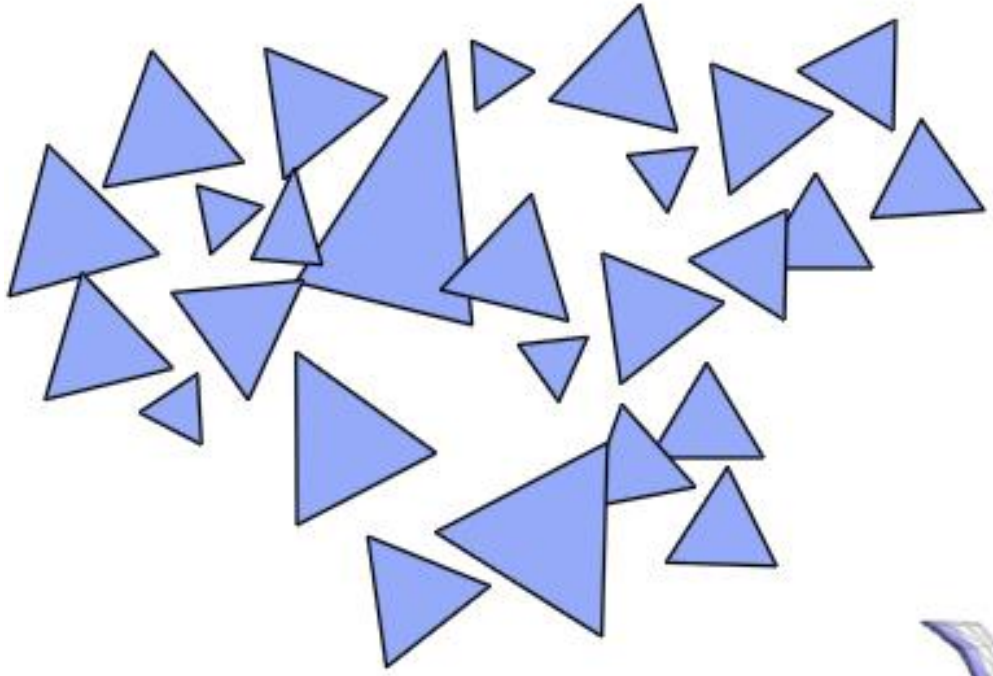
```
10    123    20    100    6    25    64    11    200    30
```

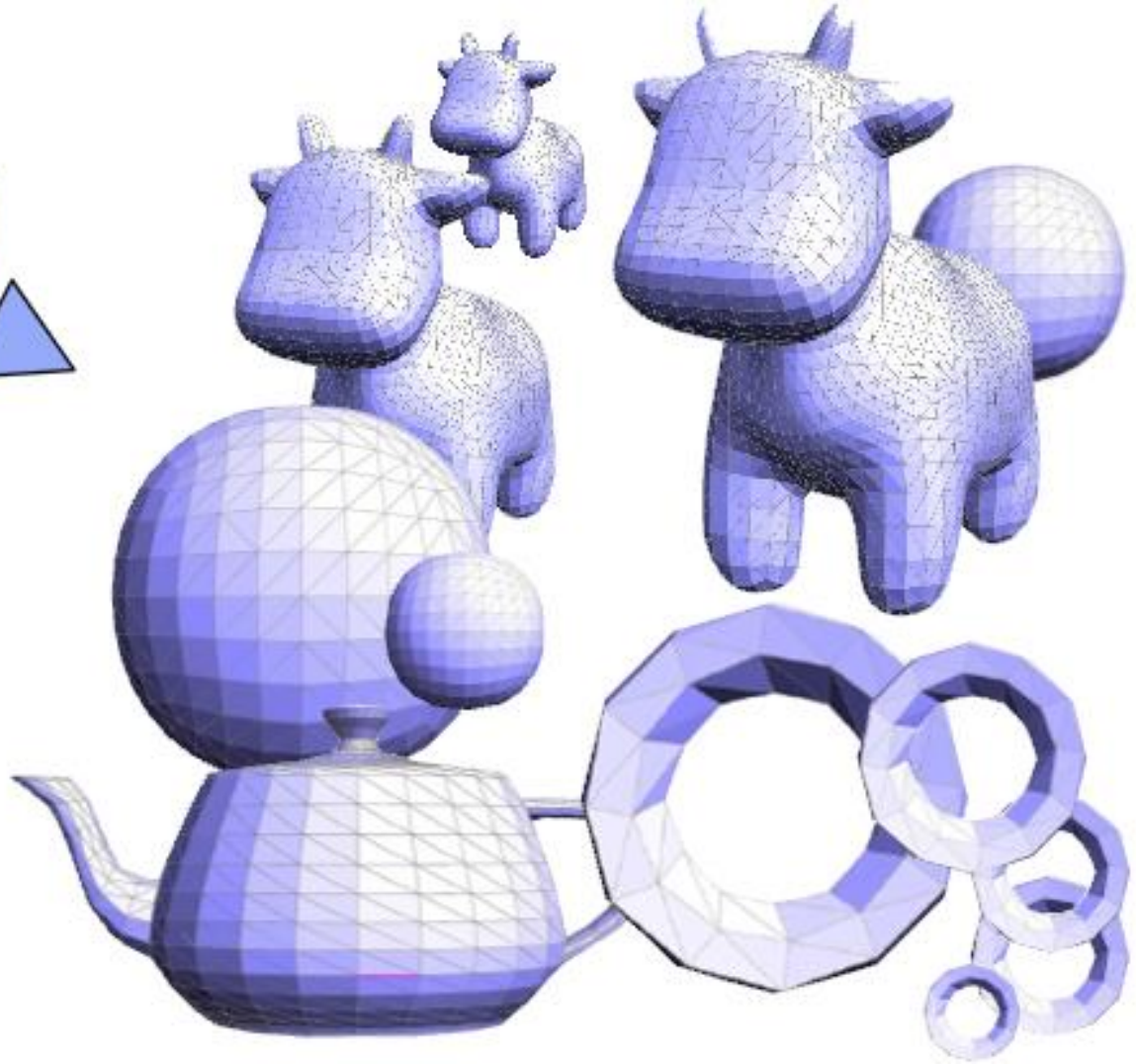
- **Example: *k*=18**
- **Sort integers:**

```
6    10    11    20    25    30    64    100    123    200
```

- **How would you perform a modified binary search?**

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





Ray-axis-aligned-box intersection

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

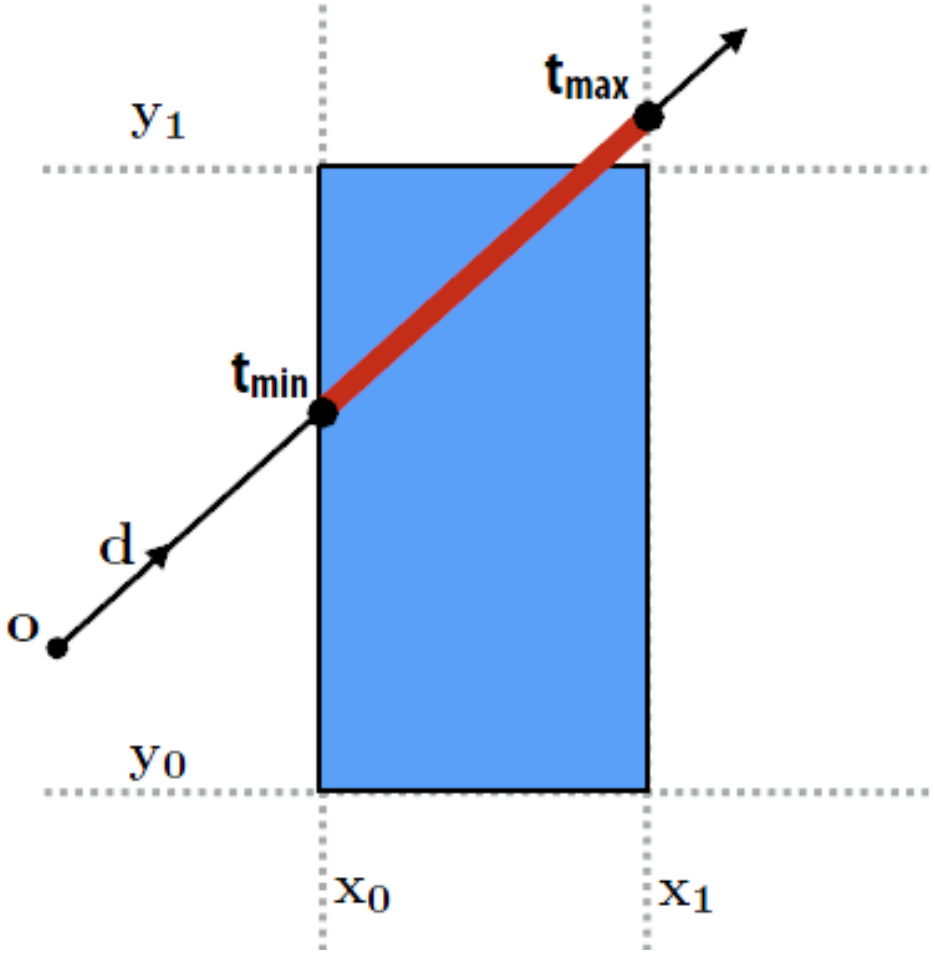


Figure shows intersections with *x*=*x*0 and *x*=*x*1 planes.

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):

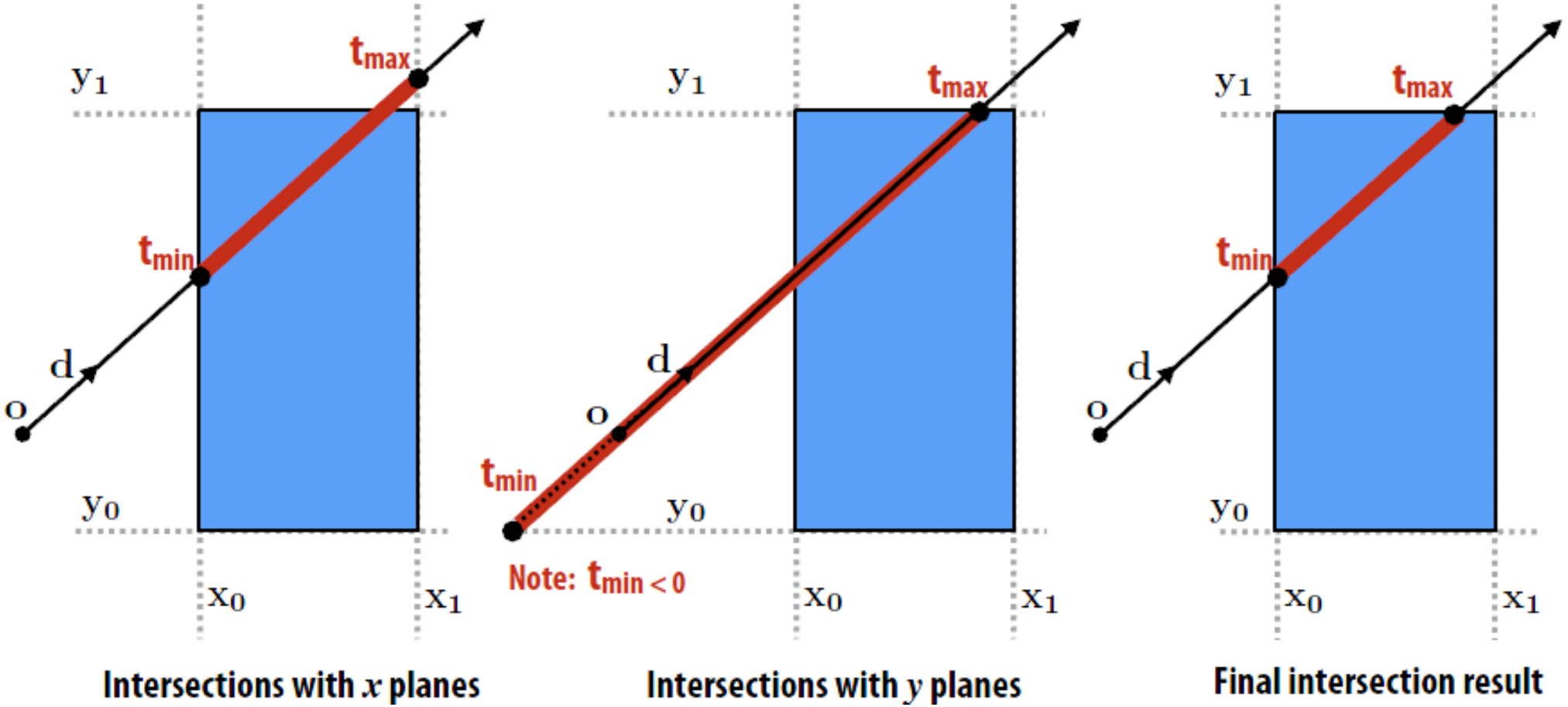
$$\mathbf{N}^{\mathbf{T}} = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$$

$$c = x_0$$

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

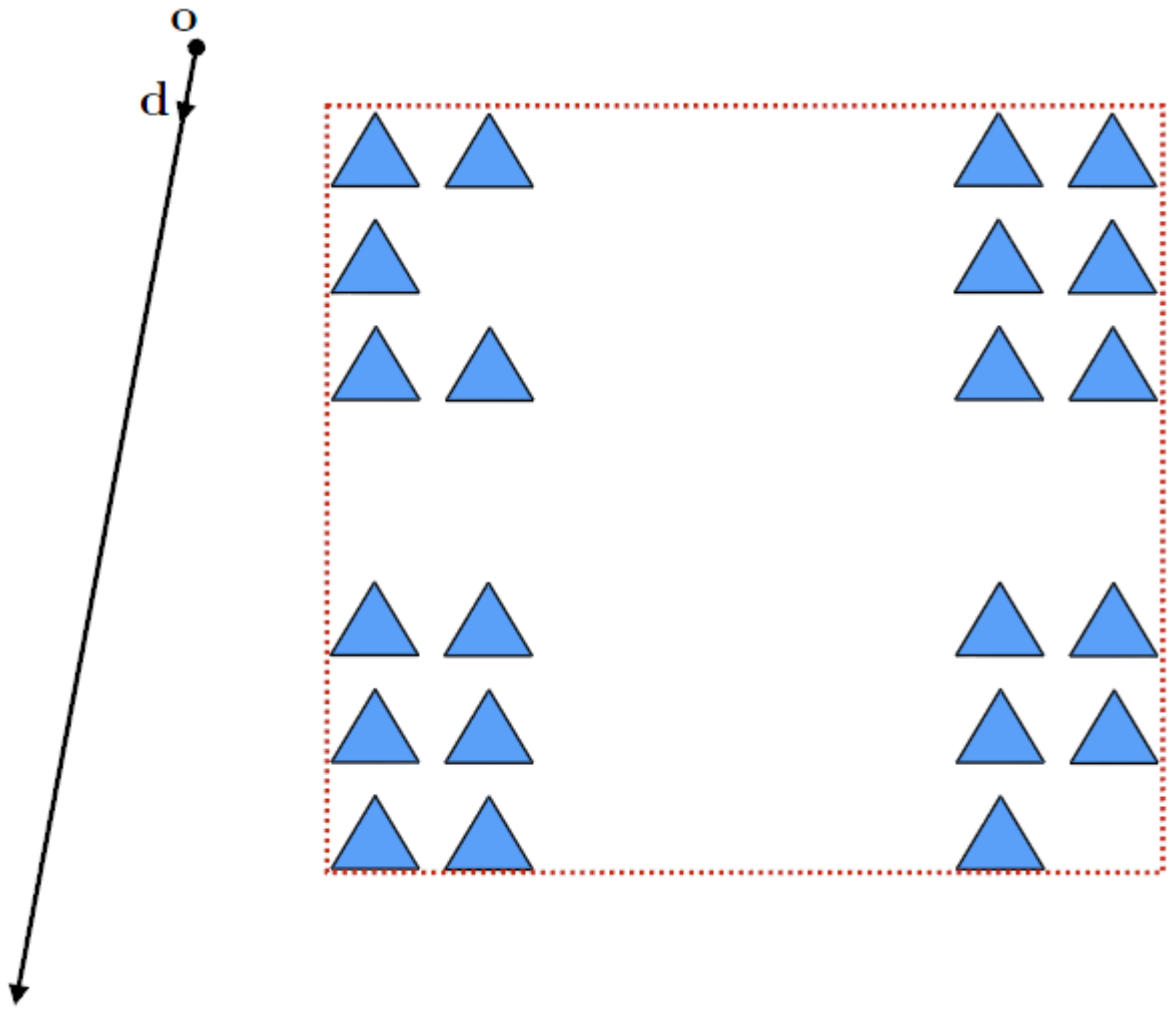
Ray-axis-aligned-box intersection

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



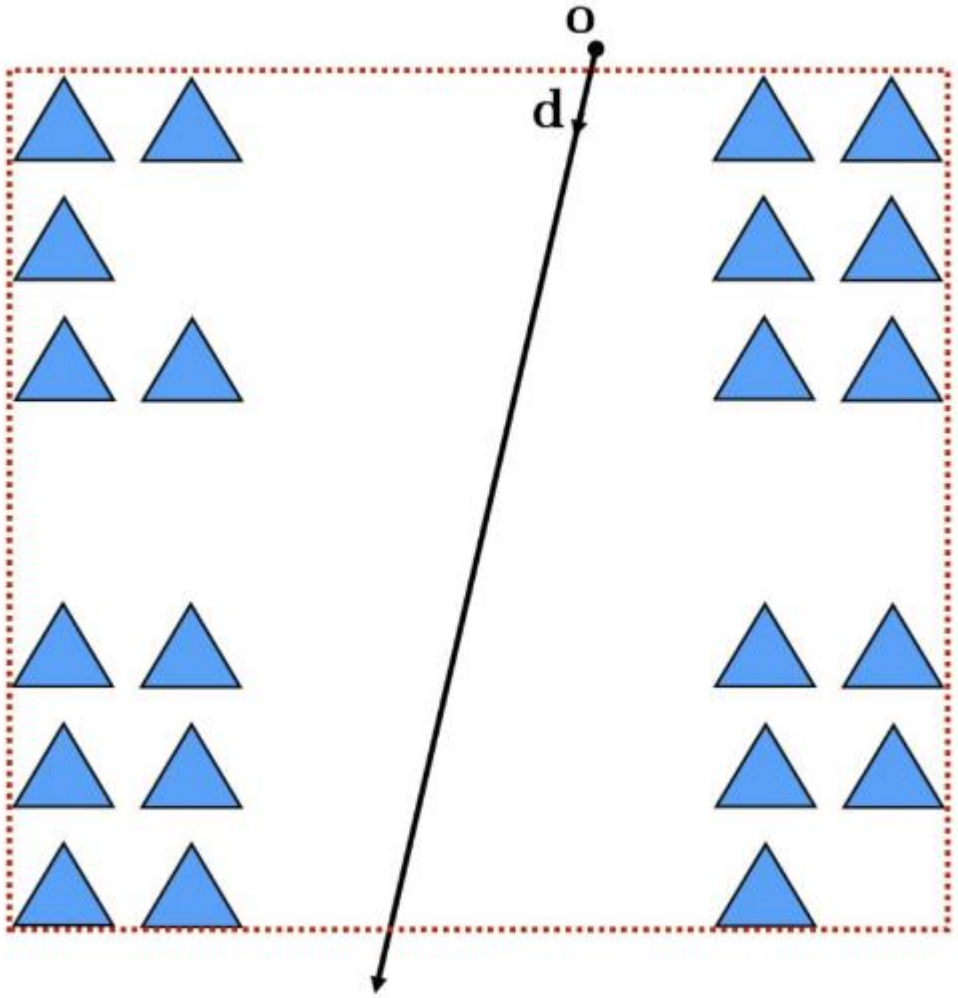
How do we know when the ray misses the box?

Simple case



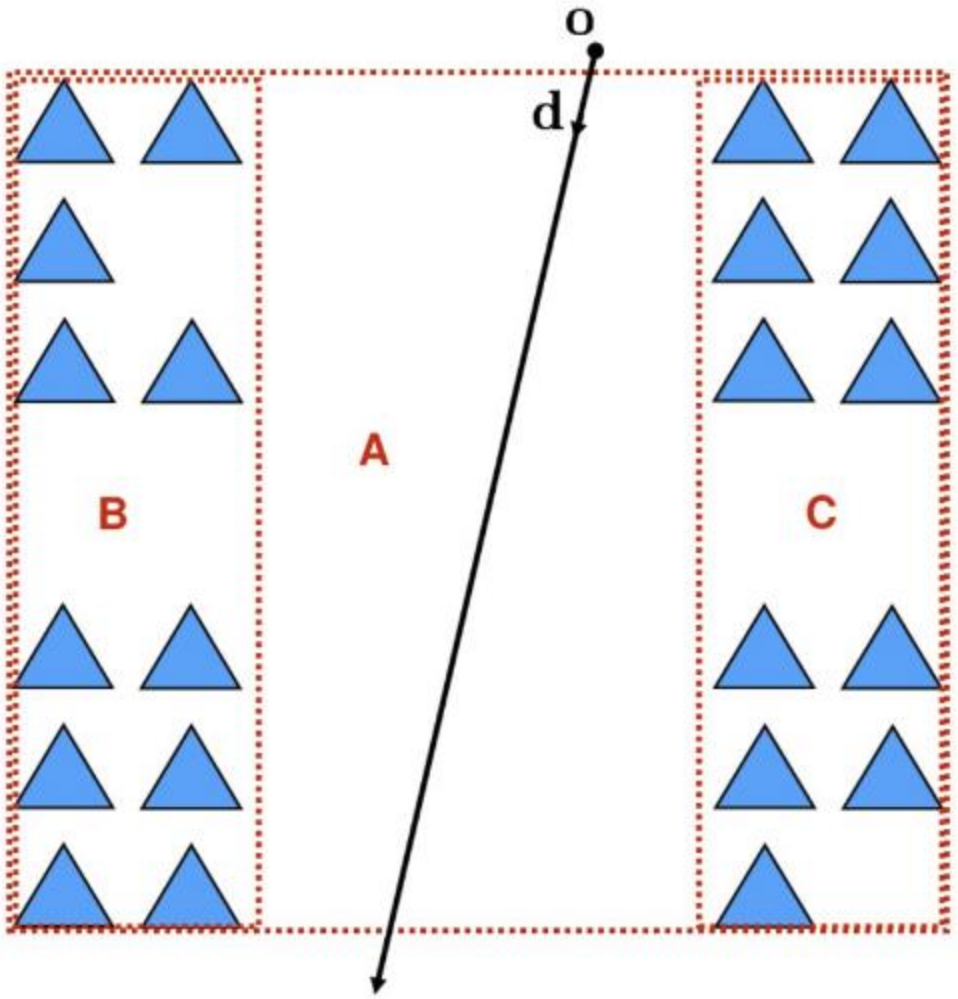
Ray misses bounding box of all primitives in scene
O(1) cost: requires 1 ray-box test

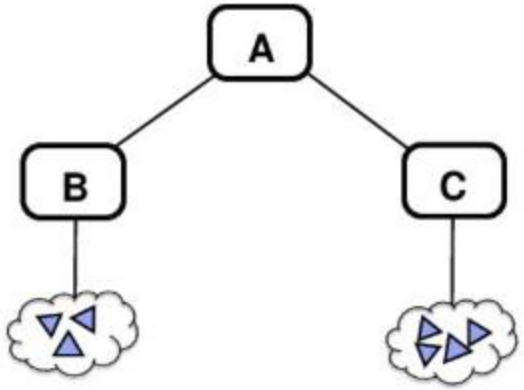
Another simple case



Ray hits bounding box, check all primitives O(N) cost :)☹

Another simple case



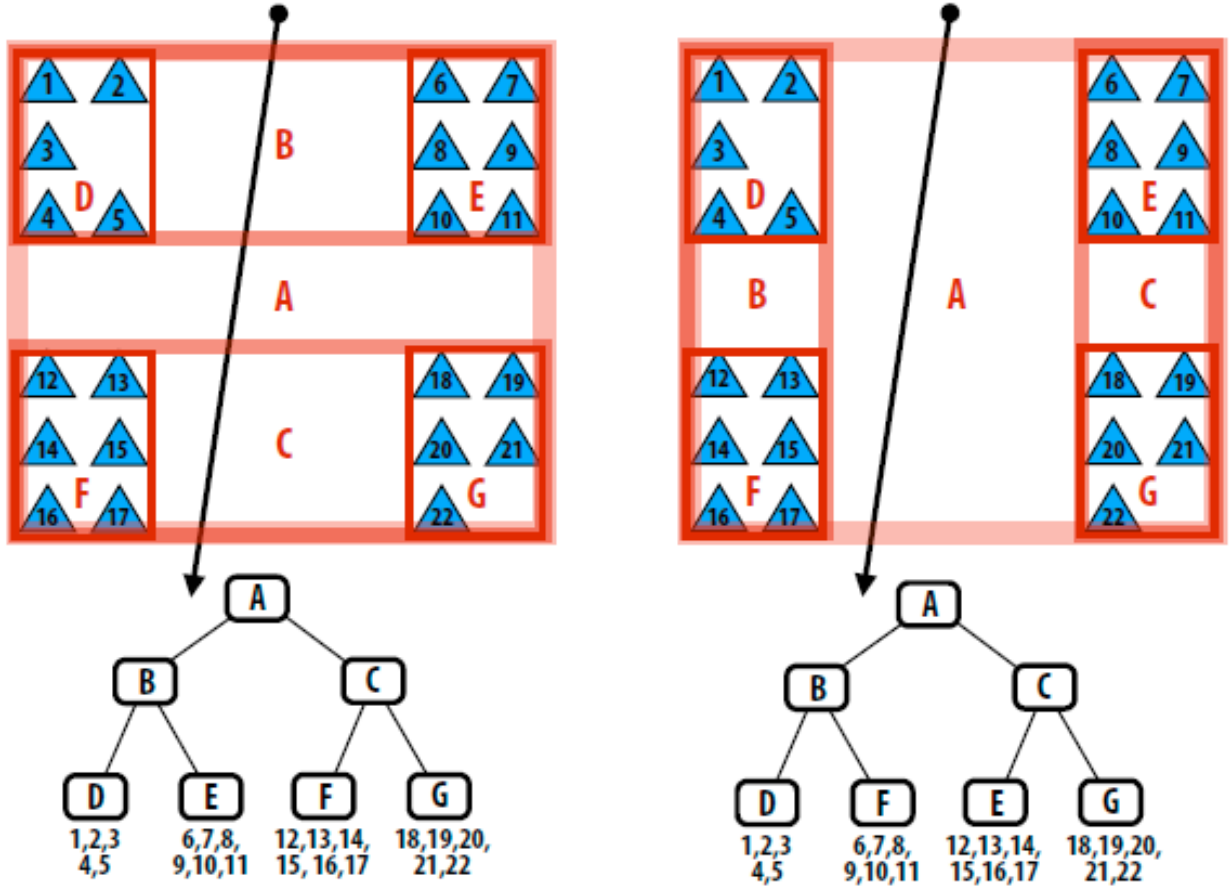


A bounding box of bounding boxes!
There is no reason to stop there!

Bounding volume hierarchy (BVH)

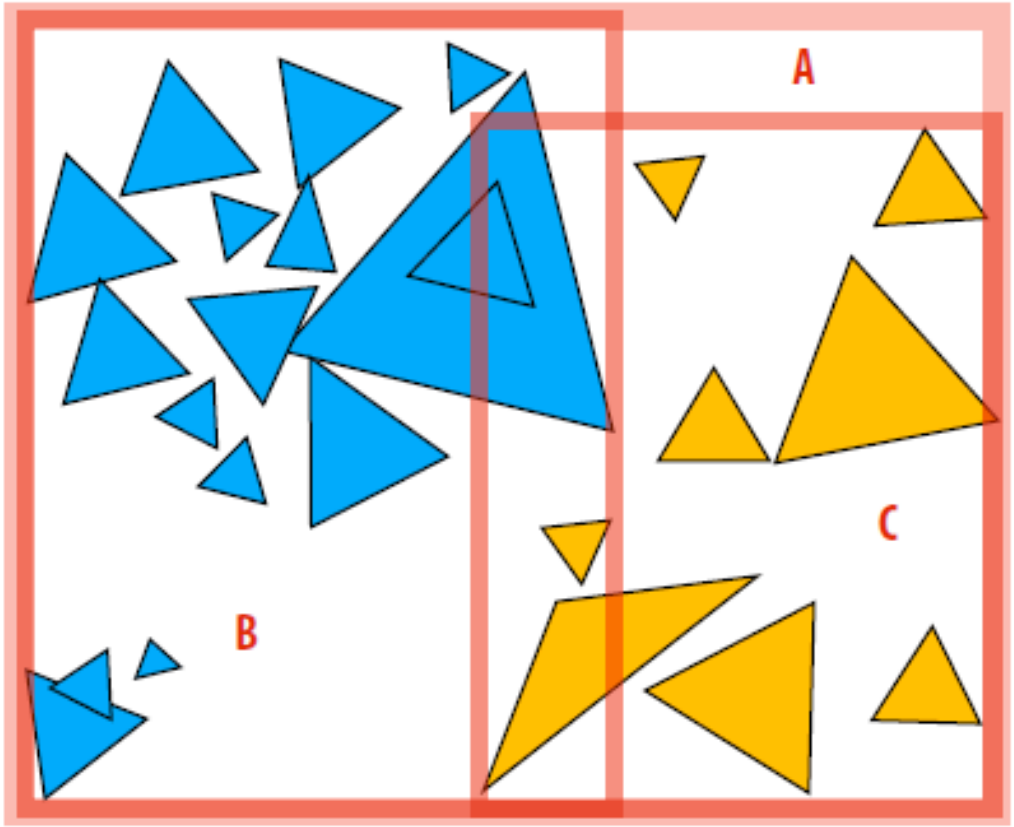
- **Interior nodes:**
  - **Represents subset of primitives in scene**
  - **Stores aggregate bounding box for all primitives in subtree**
- **Leaf nodes:**
  - **Contain list of primitives**

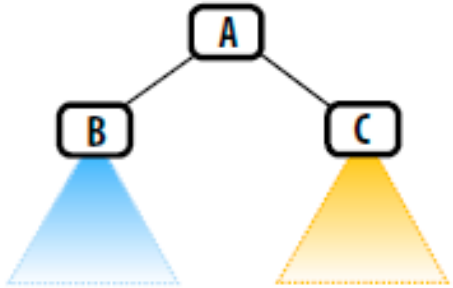
Two different BVH organizations of the same scene containing 22 primitives. Leaf node are the same.



Another BVH example

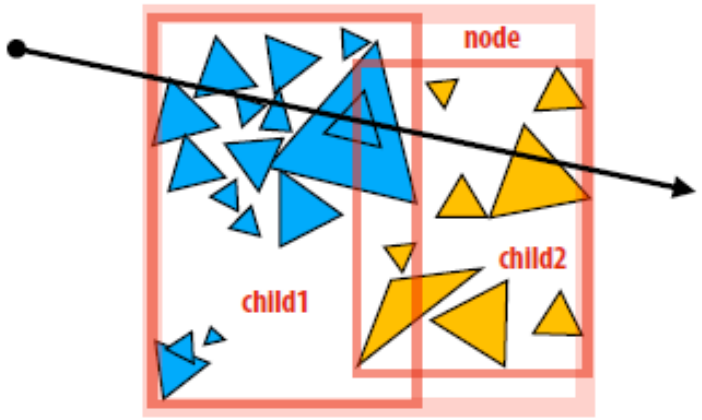
- **BVH partitions each node's primitives into disjoints sets**
  - **Note: The sets can still be overlapping in space (below: child bounding boxes may overlap in space)**





Ray-scene intersection using a BVH

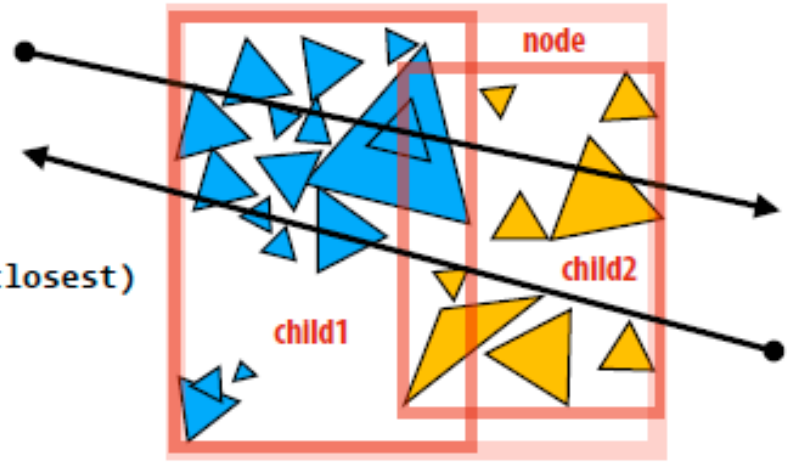
```
struct BVHNode {
   bool leaf;
   BBox bbox;
   BVHNode* child1;
   BVHNode* child2;
   Primitive* primList;
};

struct ClosestHitInfo {
   Primitive prim;
   float min_t;
};

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

   if (!intersect(ray, node->bbox) || (closest point on box is farther than closest.min_t))
      return;

   if (node->leaf) {
      for (each primitive p in node->primList) {
         (hit, t) = intersect(ray, p);
         if (hit && t < closest.min_t) {
            closest.prim = p;
            closest.min_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, ClosestHitInfo* closest)
{

   if (node->leaf) {
      for (each primitive p in node->primList) {
         (hit, t) = intersect(ray, p);
         if (hit && t < closest.min_t) {
            closest.prim = p;
            closest.min_t = t;
         }
      }
   } else {
      (hit1, min_t1) = intersect(ray, node->child1->bbox);
      (hit2, min_t2) = intersect(ray, node->child2->bbox);

      NVHNode* first = (min_t1 <= min_t2) ? child1 : child2;
      NVHNode* second = (min_t1 <= min_t2) ? child2 : child1;

      find_closest_hit(ray, first, closest);
      if (second child's min_t is closer than closest.min_t)
         find_closest_hit(ray, second, closest);
   }
}
```

“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) );
   }
}
```

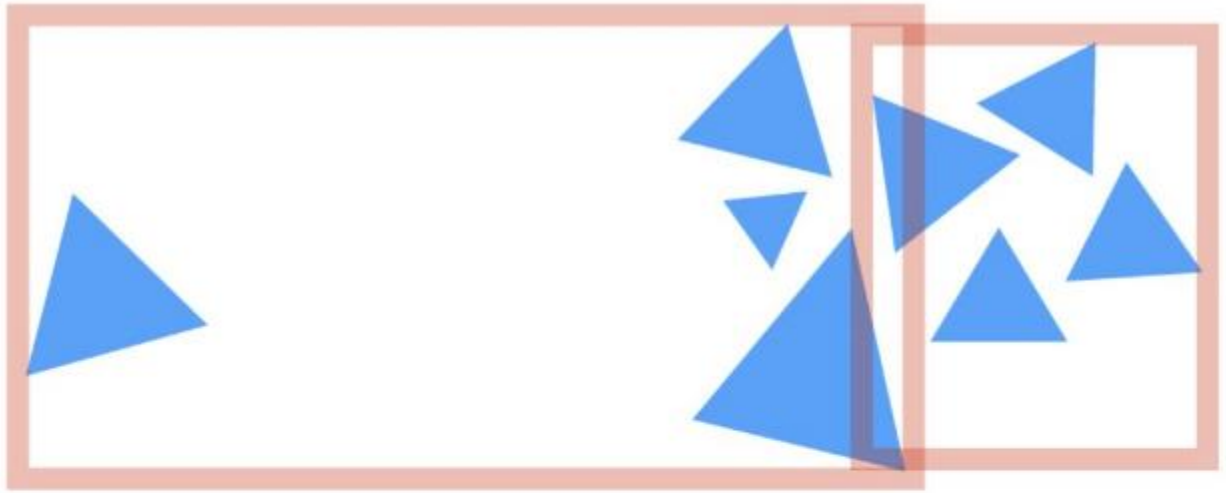
Interesting question of which child to enter first. How might you make a good decision?

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

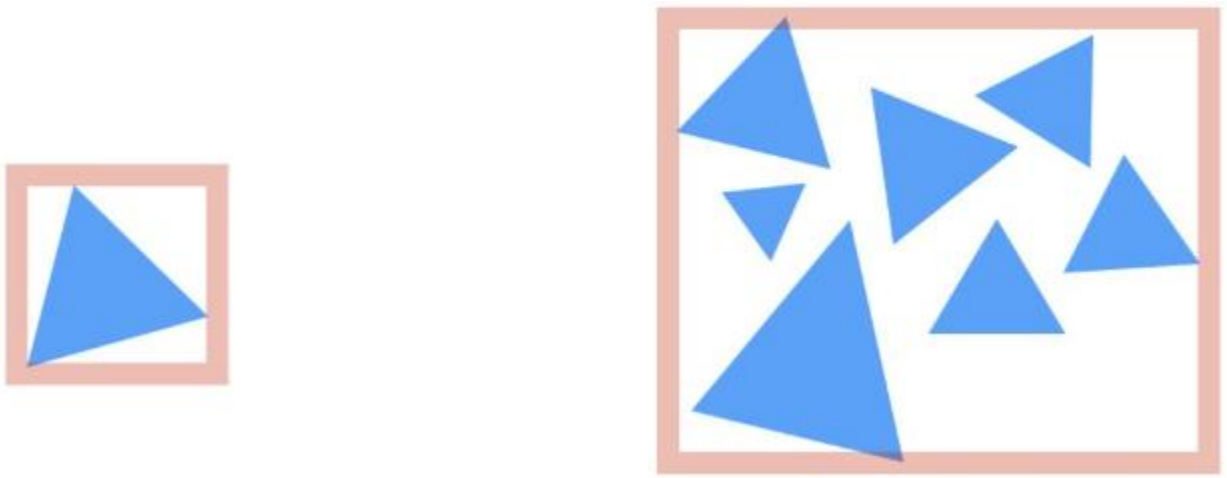
- Many ways are there to partition N primitives into to groups?

- How do we build a high-quality BVH?

Intuition about a “good” partition?



Partition into child nodes with equal numbers of primitives



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:

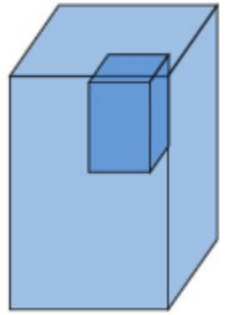
$$C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}}$$

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

Estimating probabilities

- **For 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):**

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

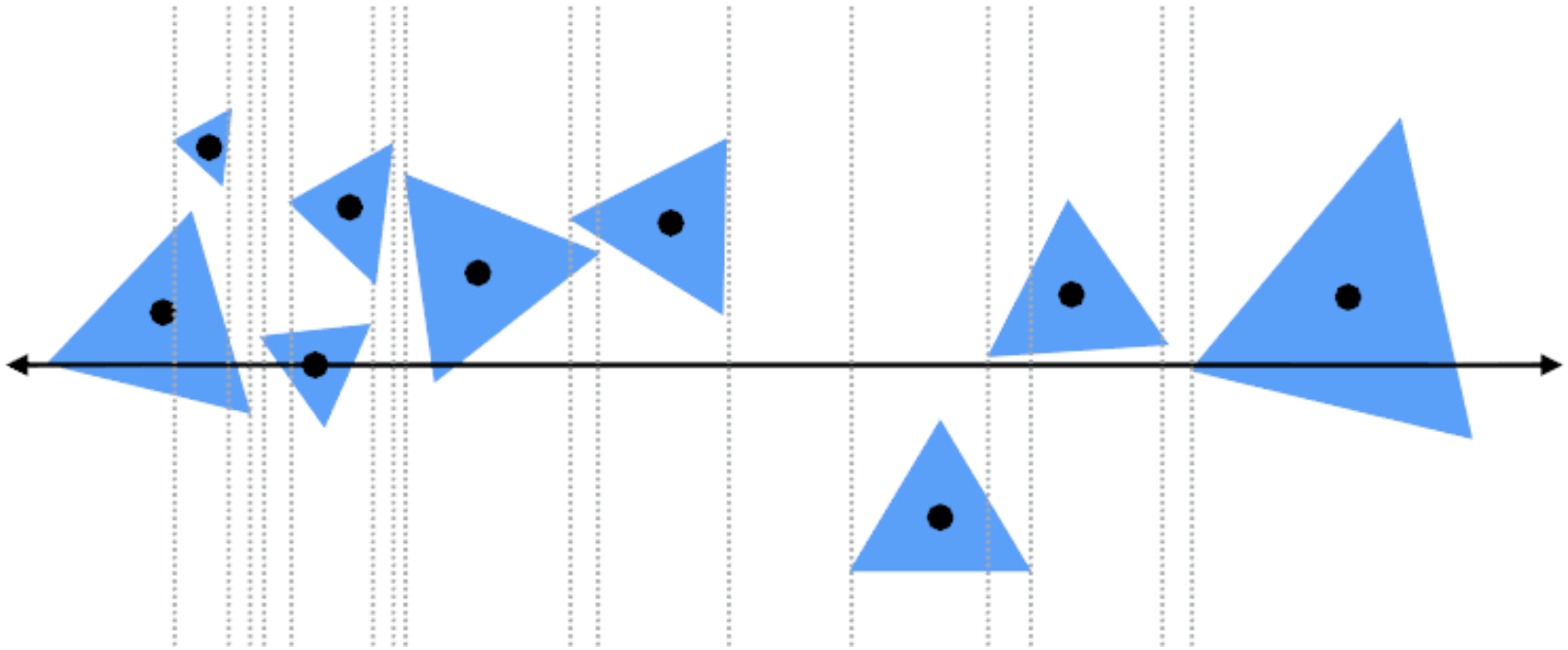
Assumptions of the SAH (may not hold in practice):

Rays are randomly distributed

Rays are not occluded

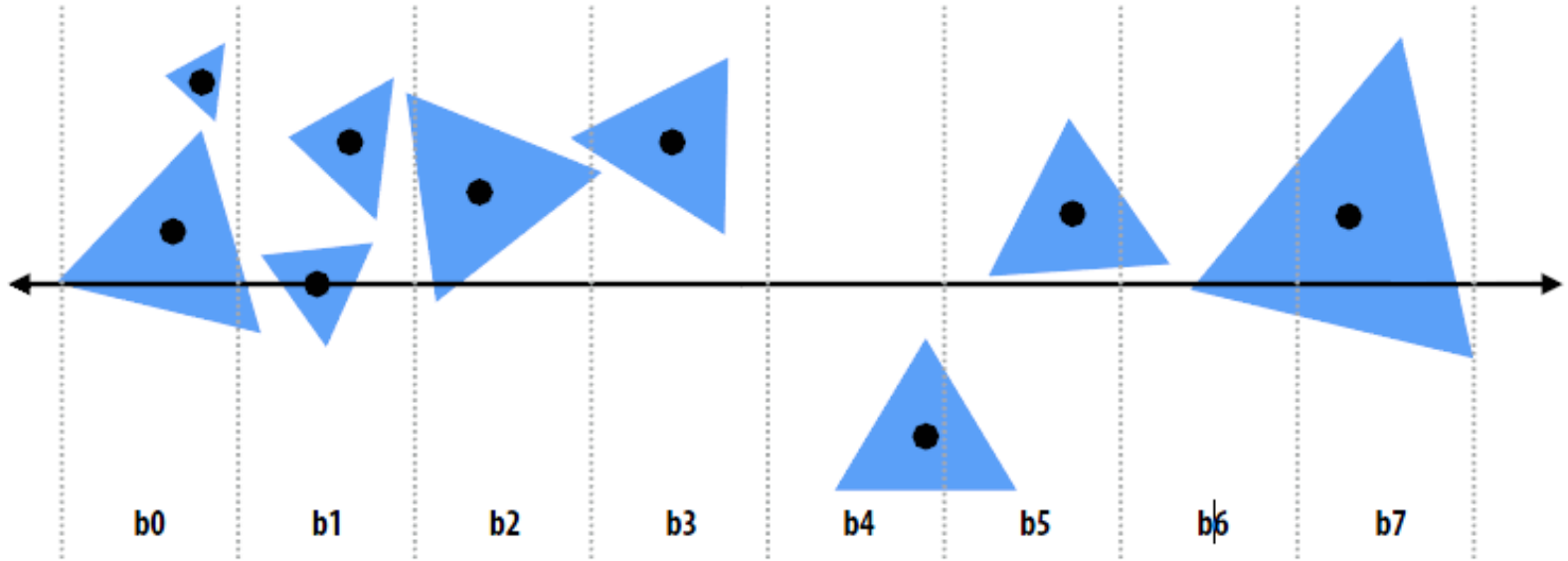
Implementing partitions

- **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**



Efficiently implementing partitioning

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

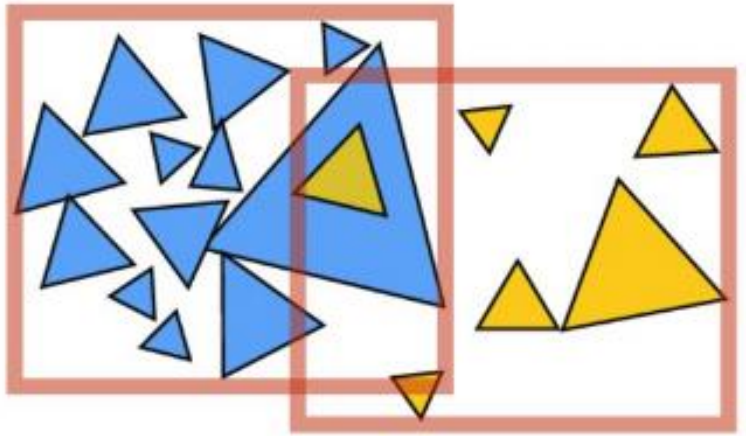



```
For each axis: x,y,z:
   initialize buckets
   For each primitive p in node:
      b = compute_bucket(p.centroid)
      b.bbox.union(p.bbox);
      b.prim_count++;
   For each of the B-1 possible partitioning planes evaluate SAH
Execute lowest cost partitioning found (or make node a leaf)
```

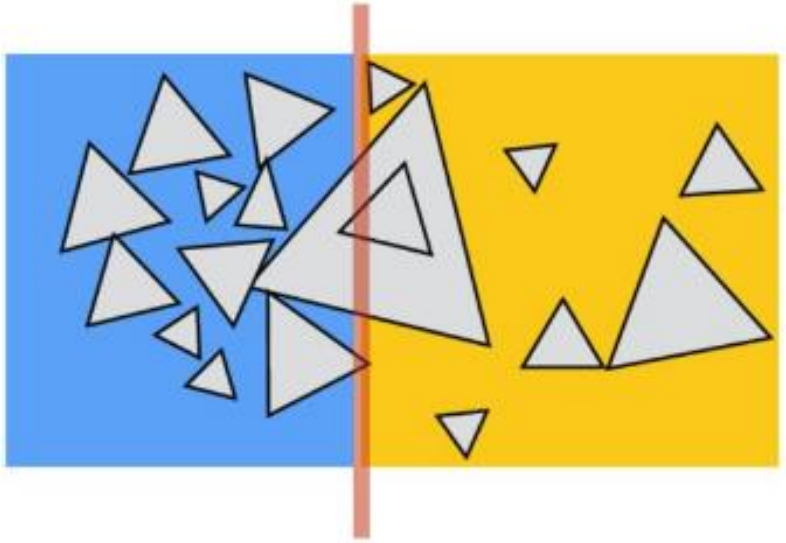
$$C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}}$$

Primitive-partitioning acceleration structures vs. space-partitioning structures

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

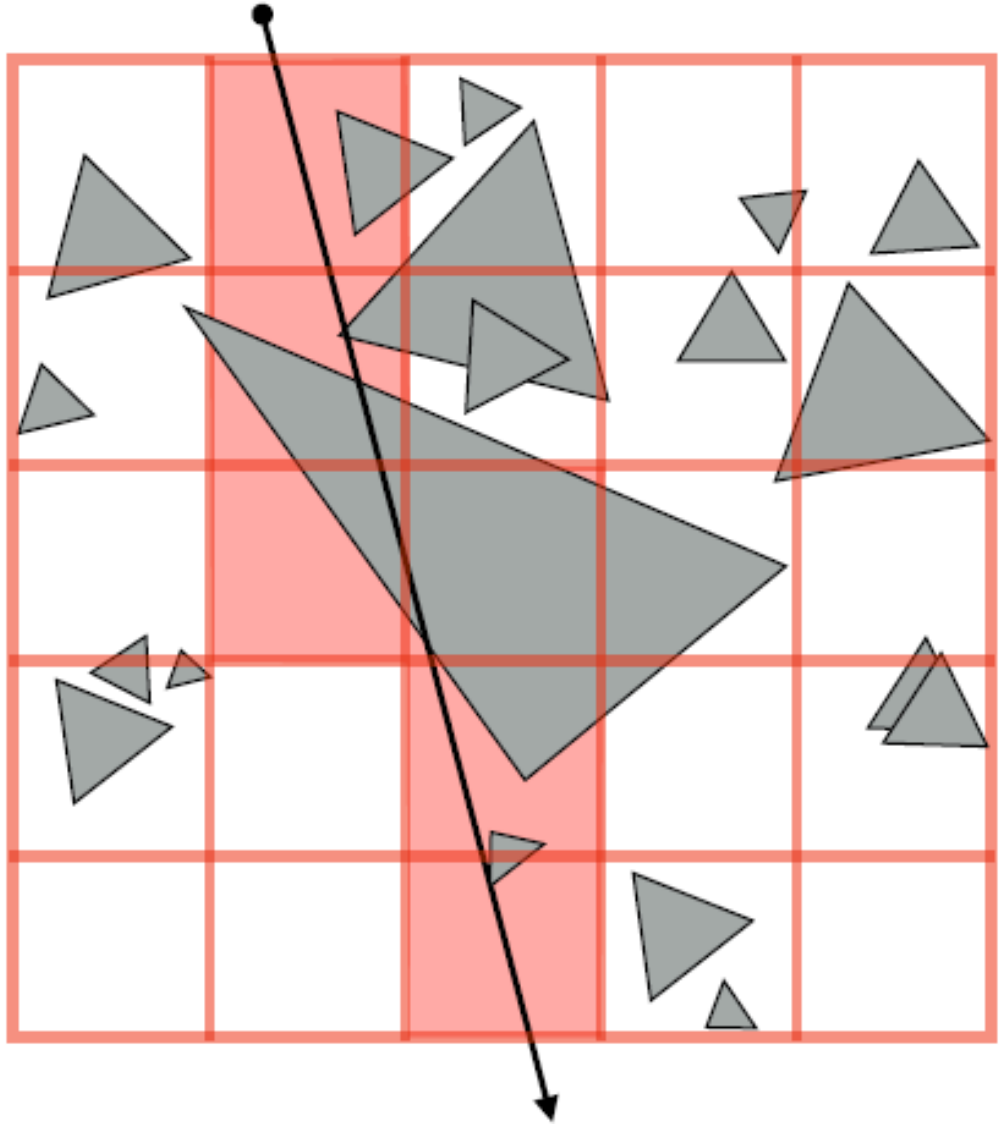


- **Space-partitioning (grid, K-D tree) partitions space into disjoint regions (primitives may be contained in multiple regions of space)**

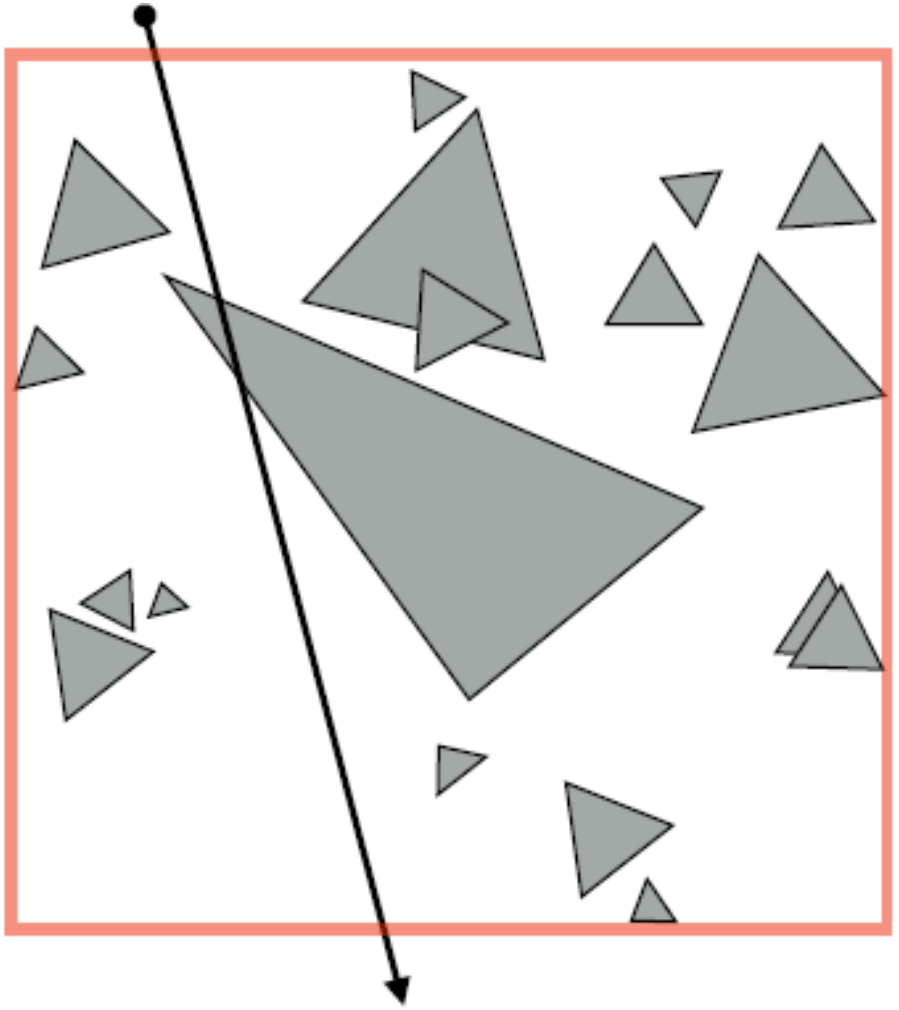


Uniform grid

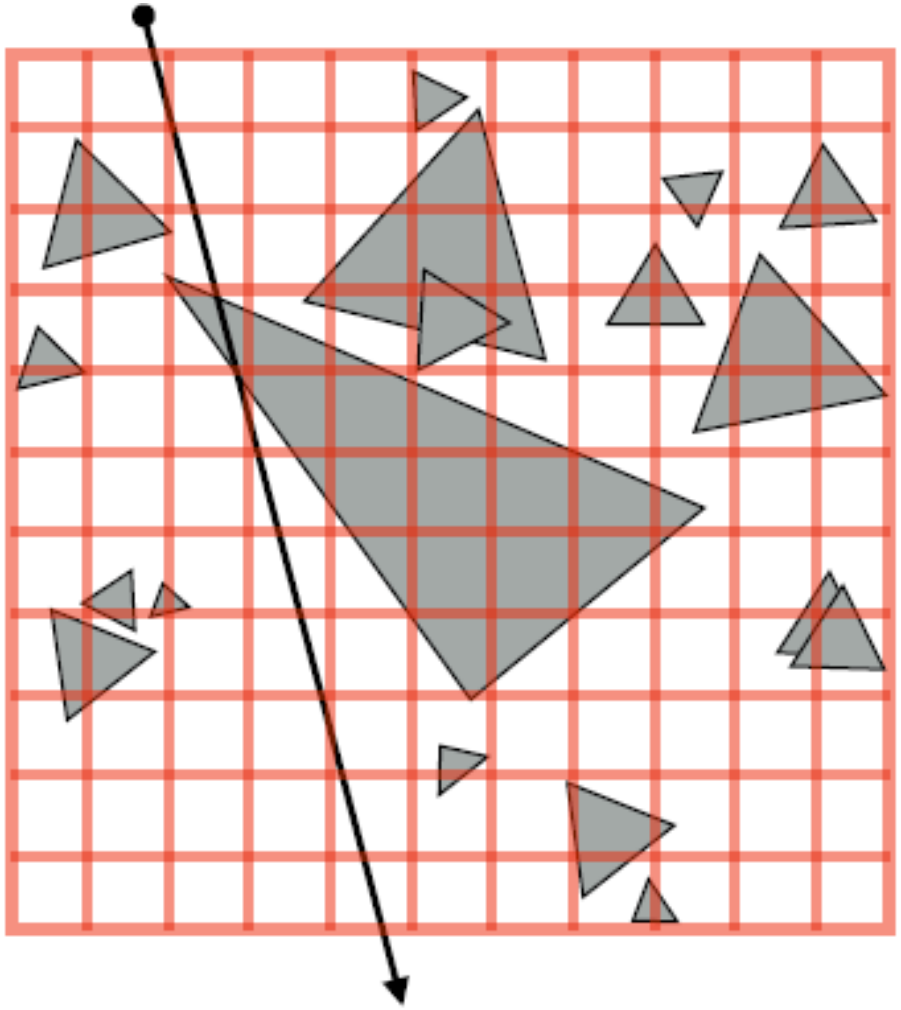
- Partition space into equal sized volumes (“voxels”)
- Each grid cell contains primitives that overlap voxel. (very cheap to construct acceleration structure)
- Walk ray through volume in order
  - Very efficient implementation possible (think: 3D line rasterization)
  - Only consider intersection with primitives in voxels the ray intersects



What should the grid resolution be?



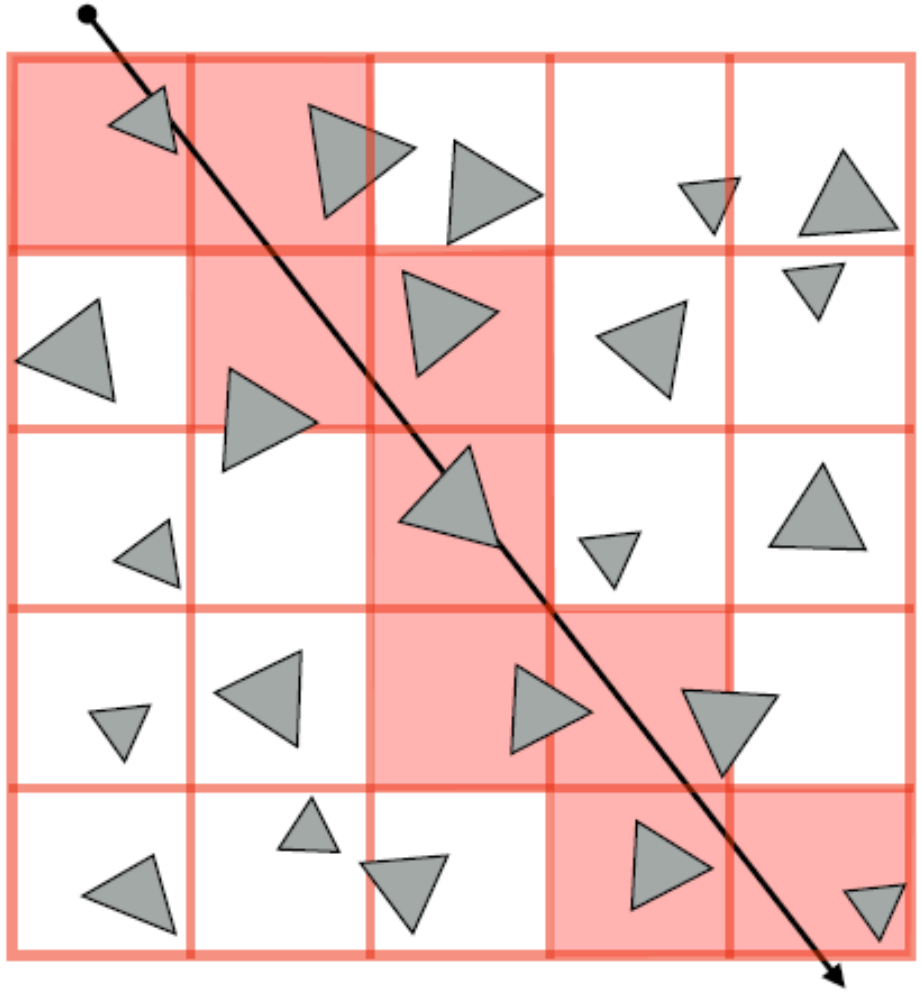
Too few grids cell: degenerates to brute-force approach



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

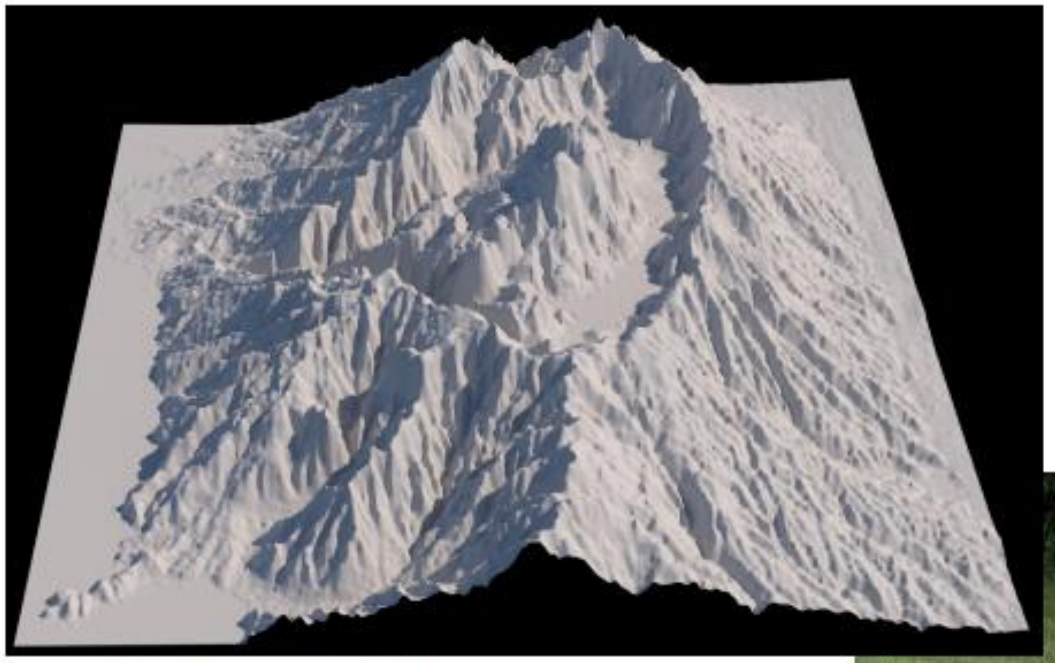
Heuristic

- **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:

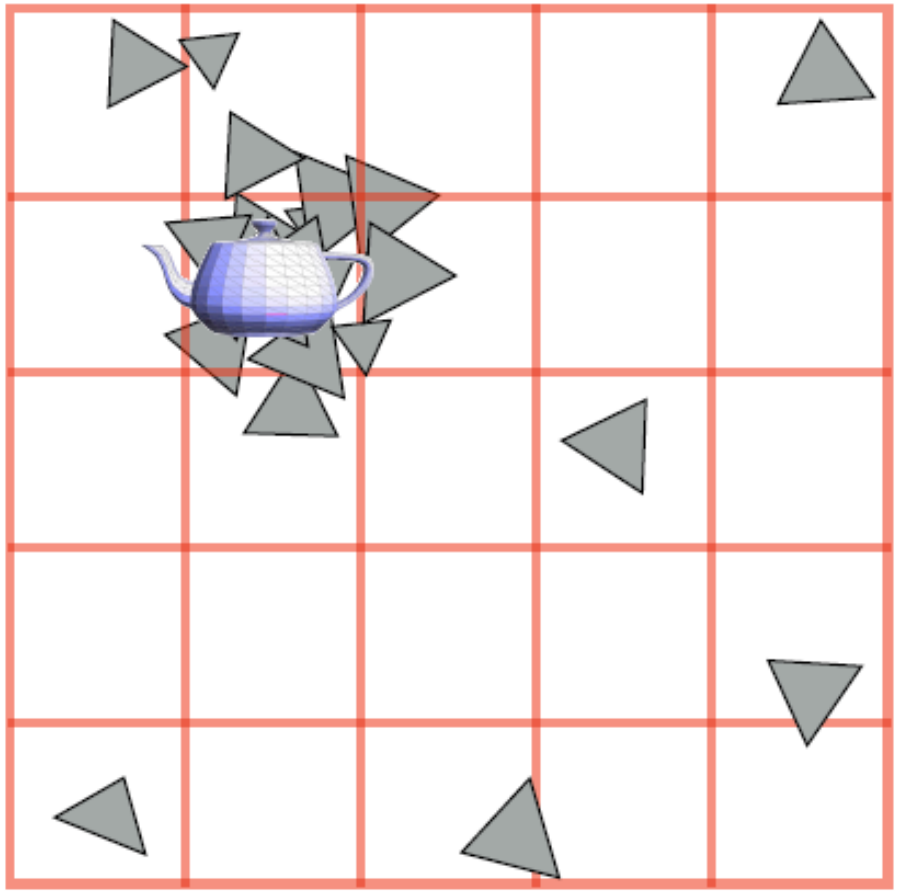
[Image credit: Misuba Renderer]

Grass:

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

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

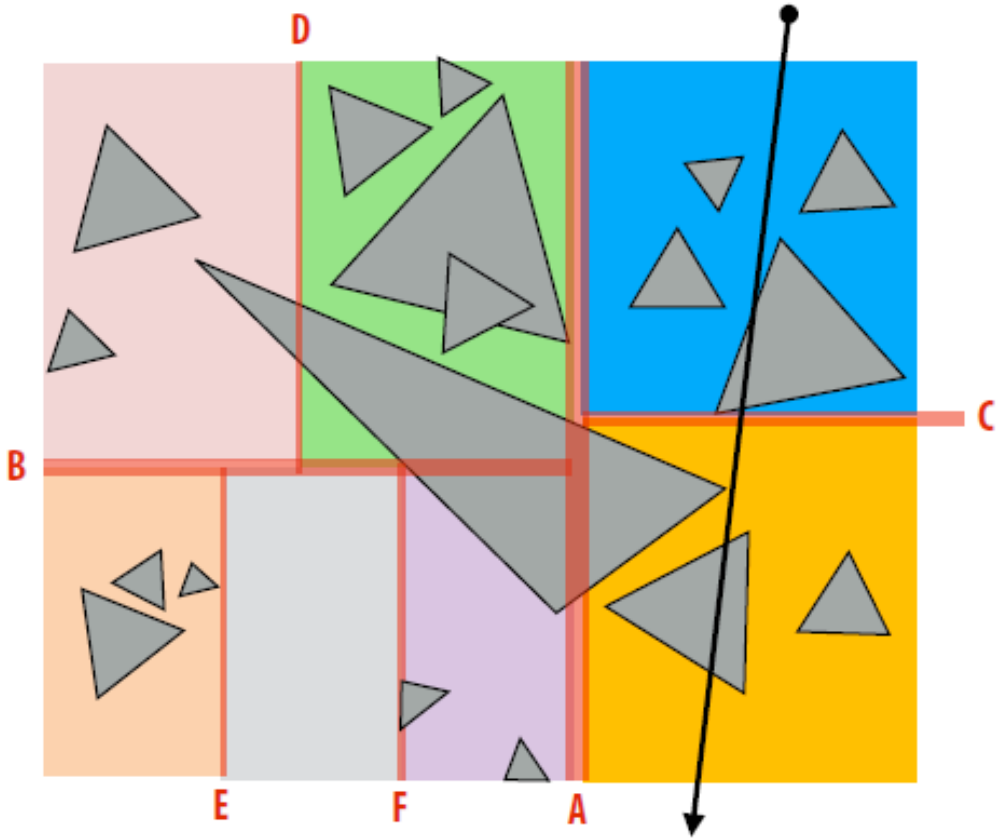
- “Teapot in a stadium problem”
- Scene has large spatial extent.
- Contains a high-resolution object that
- has small spatial extent (ends up in one
- grid cell)

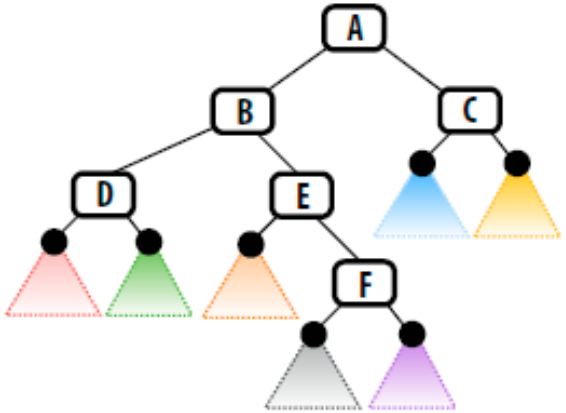


Non-uniform distribution of geometric detail requires adaptive grids

K-D tree

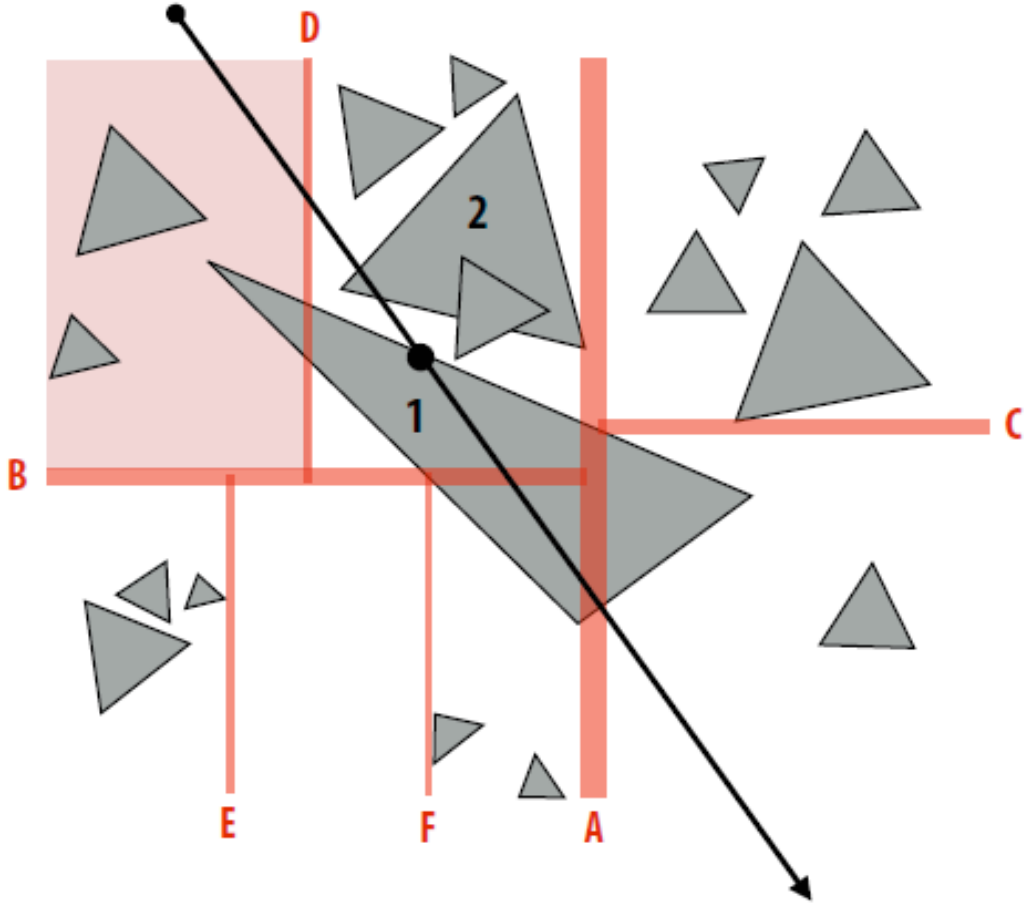
- **Recursively partition space via axis-aligned partitioning planes**
  - **Interior nodes correspond to spatial splits (still correspond to spatial volume)**
  - **Node traversal can proceed in front-to-back order (unlike BVH, can terminate search after first hit is found).**

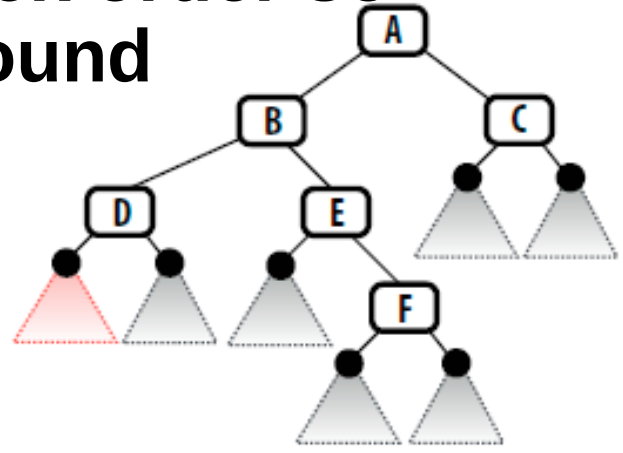




Challenge: objects overlap multiple nodes

- **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.

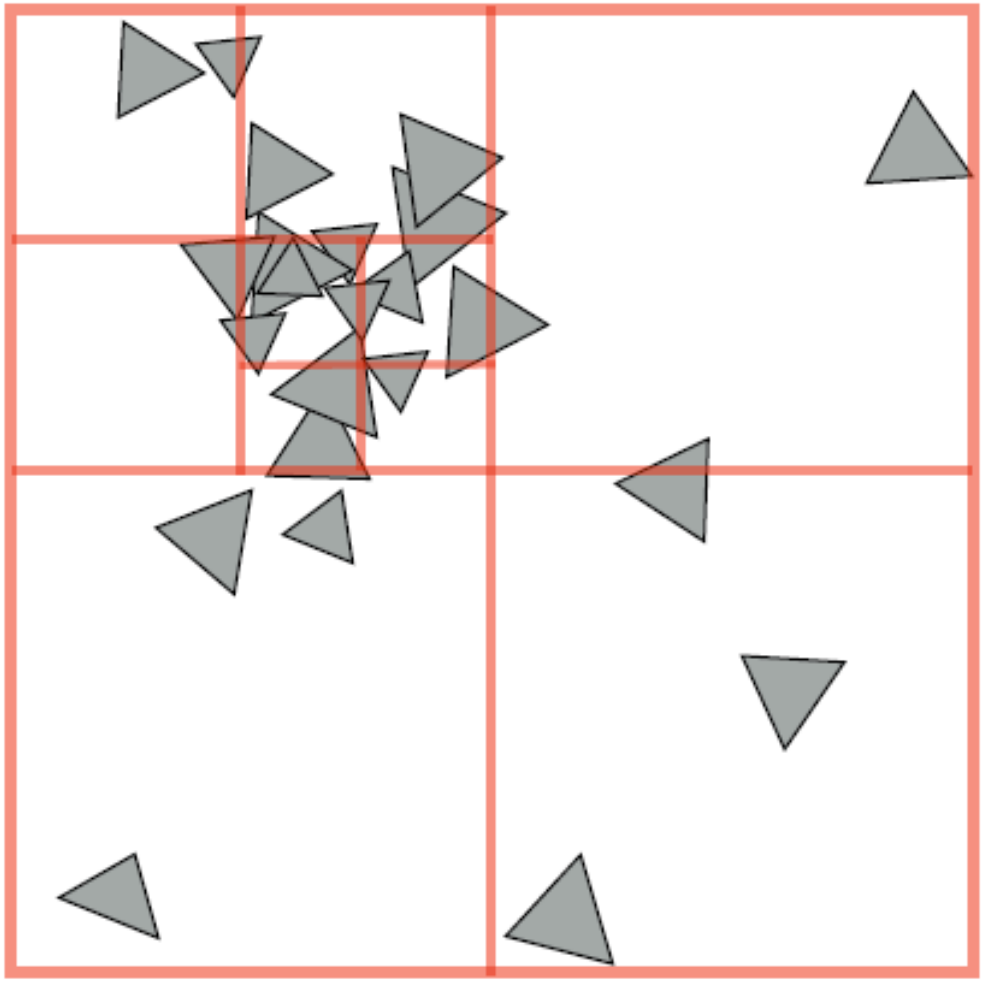
But intersection with triangle 2 is closer! (Haven't traversed to that node yet)

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

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

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)

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

Thank you