Computer Graphics

Acceleration Data Structures

Prof. Dr. Markus Gross

grossm@inf.ethz.ch

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

computer graphics laboratory

Disney RESEARCH STUDIOS

The Quest for Realism

- Realism through geometric complexity

The Quest for Realism

- Realism through geometric complexity

The Quest for Realism

- Realism through geometric complexity

Overview

Acceleration techniques

- Fewer intersection computations
  - uniform grids
  - binary space partition (BSP-tree), KD-tree, Octree
  - Bounding volume hierarchies (BVH)
- Fewer rays
  - early ray termination
  - adaptive sampling
- Generalized rays
  - beam tracing
  - cone tracing

Overview

Acceleration techniques

Fewer intersection computations

- uniform grids
- binary space partition (BSP-tree), KD-tree, Octree
- Bounding volume hierarchies (BVH)

Fewer rays

- early ray termination
- adaptive sampling

Generalized rays

- beam tracing
- cone tracing

Ray Tracing Acceleration

- Ray-surface intersection is at the core of every ray tracing algorithm

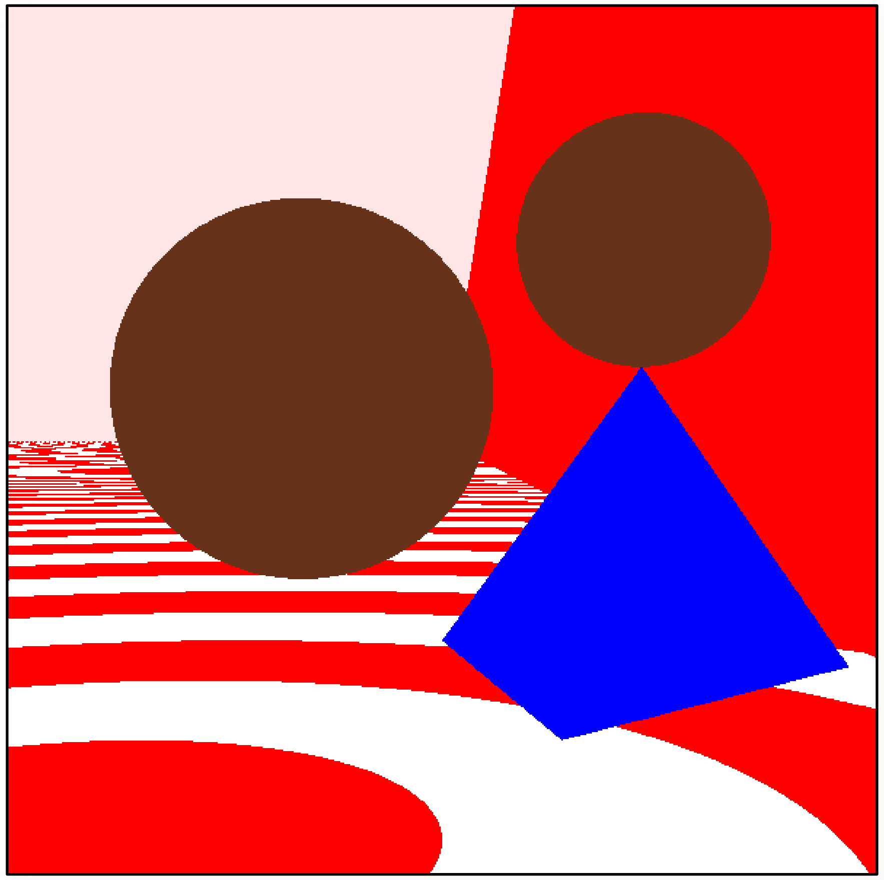
- Brute force approach
  - intersect every ray with every primitive
  - many unnecessary ray-surface intersection tests

Oliver Deussen, Univerity of Konstanz

Ray Tracing Cost

- "the time required to compute the intersections of rays and surfaces is over 95 percent"
  —Whitted 1980
- Cost = $O(N_x \cdot N_y \cdot N_o)$
  - (number of pixels) * (number of objects)
  - Assumes 1 ray per pixel
- Example: 1000x1000 image of a scene with 1000 triangles
  - Cost is (at least) 10^9 ray-triangle intersections
- Typically measured per ray:
  - Naive: $O(N_o)$ - linear with number of objects

$O(N_o)$ Ray Tracing (The Problem)



8 primitives → 3 seconds

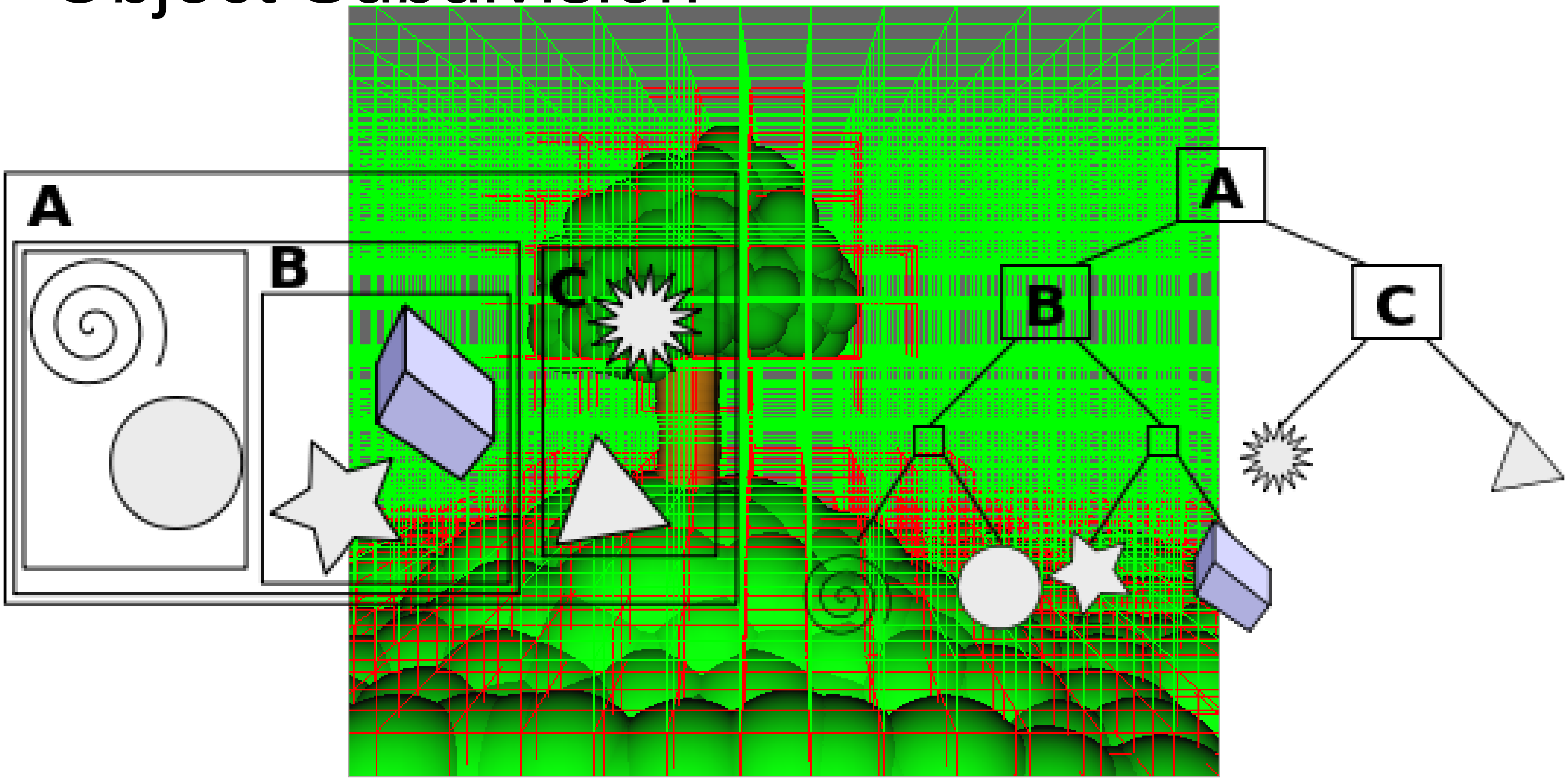
50K trees each with 1M polygons = 50B polygons → **594 years!**

Sub-linear Ray Tracing

50K trees each with 1M polygons = 50B polygons → **11 minutes**
300,000,000x speedup!

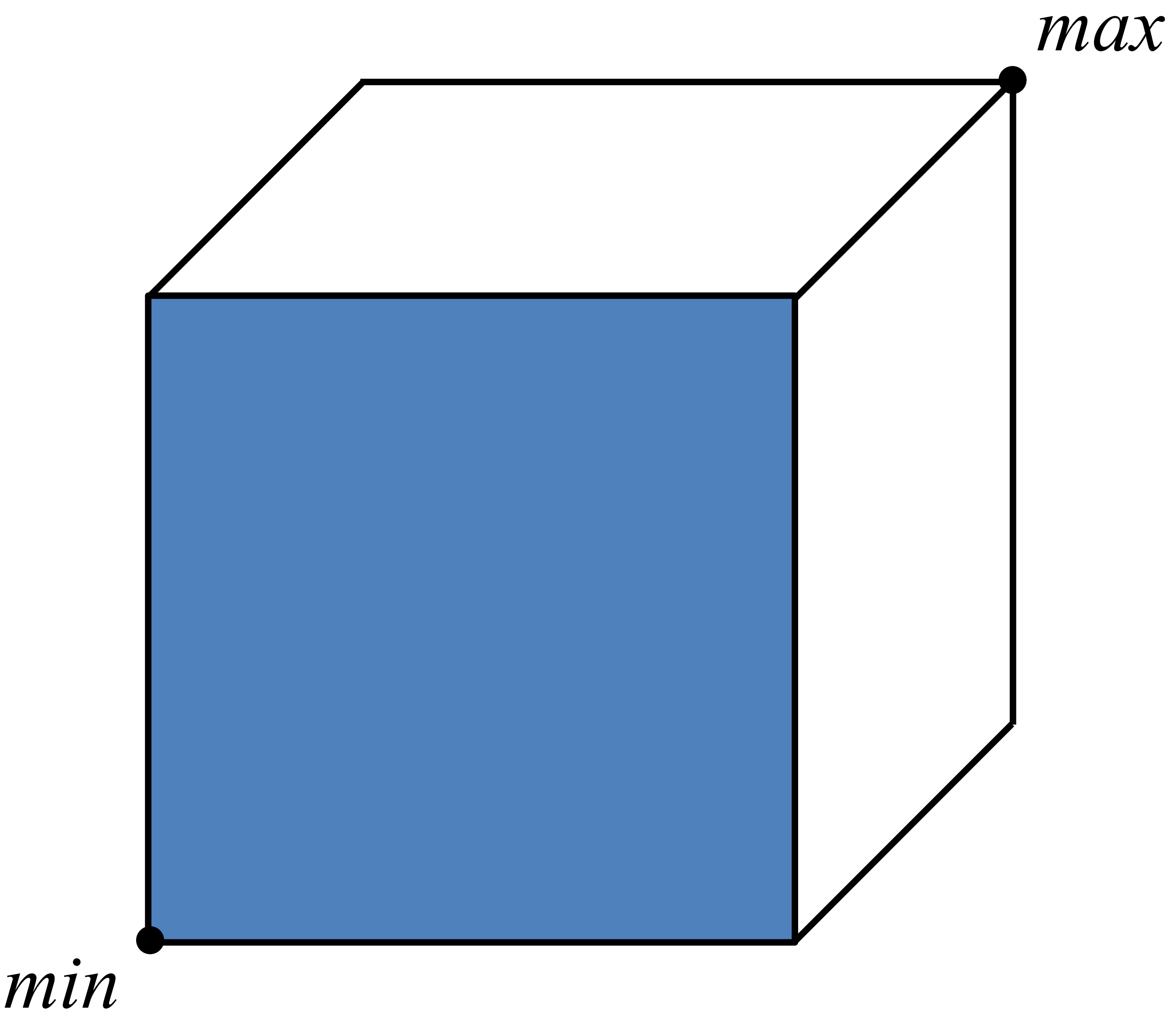
Acceleration Techniques

- Spatial Subdivision
- Object Subdivision



Axis Aligned Bounding Boxes

```
struct AABB
{
    Vector3 min;
    Vector3 max;
};
```



Ray-AABB Intersection

- Intersection of slabs

$$\mathbf{o}_x + t_{x1}\mathbf{d}_x = x_{min}$$

$$\mathbf{o}_x + t_{x2}\mathbf{d}_x = x_{max}$$

$$\downarrow$$

x slabs: solve for t_{x1}, t_{x2}

$$t_{x1} = \frac{x_{min} - \mathbf{o}_x}{\mathbf{d}_x}, t_{x2} = \frac{x_{max} - \mathbf{o}_x}{\mathbf{d}_x}$$

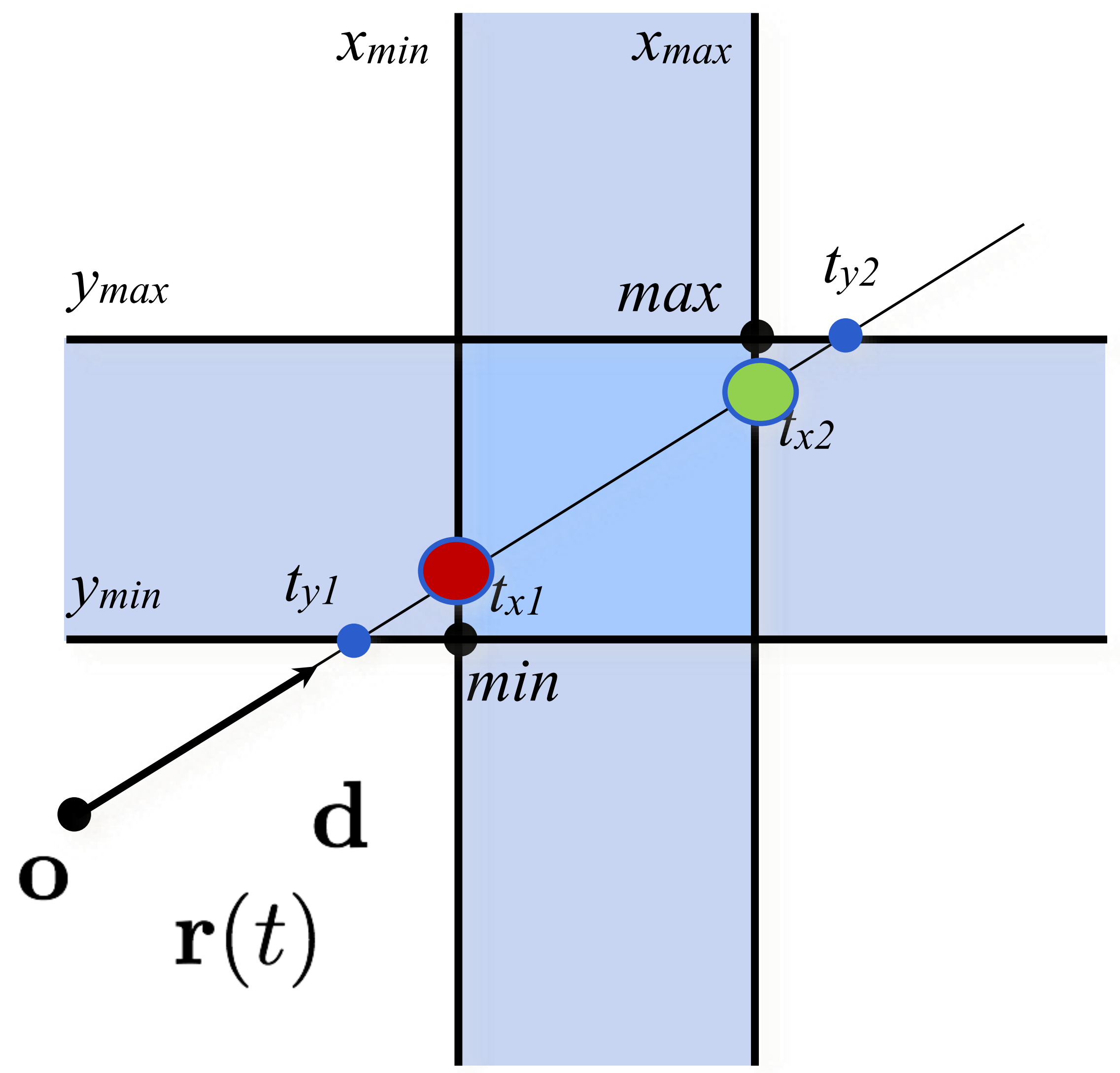
if $t_{x1} > t_{x2}$: swap(t_{x1}, t_{x2})

repeat for : $t_{y1}, t_{y2}, t_{z1}, t_{z2}$

$\boxed{t_{min}} = \max(t_{x1}, t_{y1}, t_{z1})$

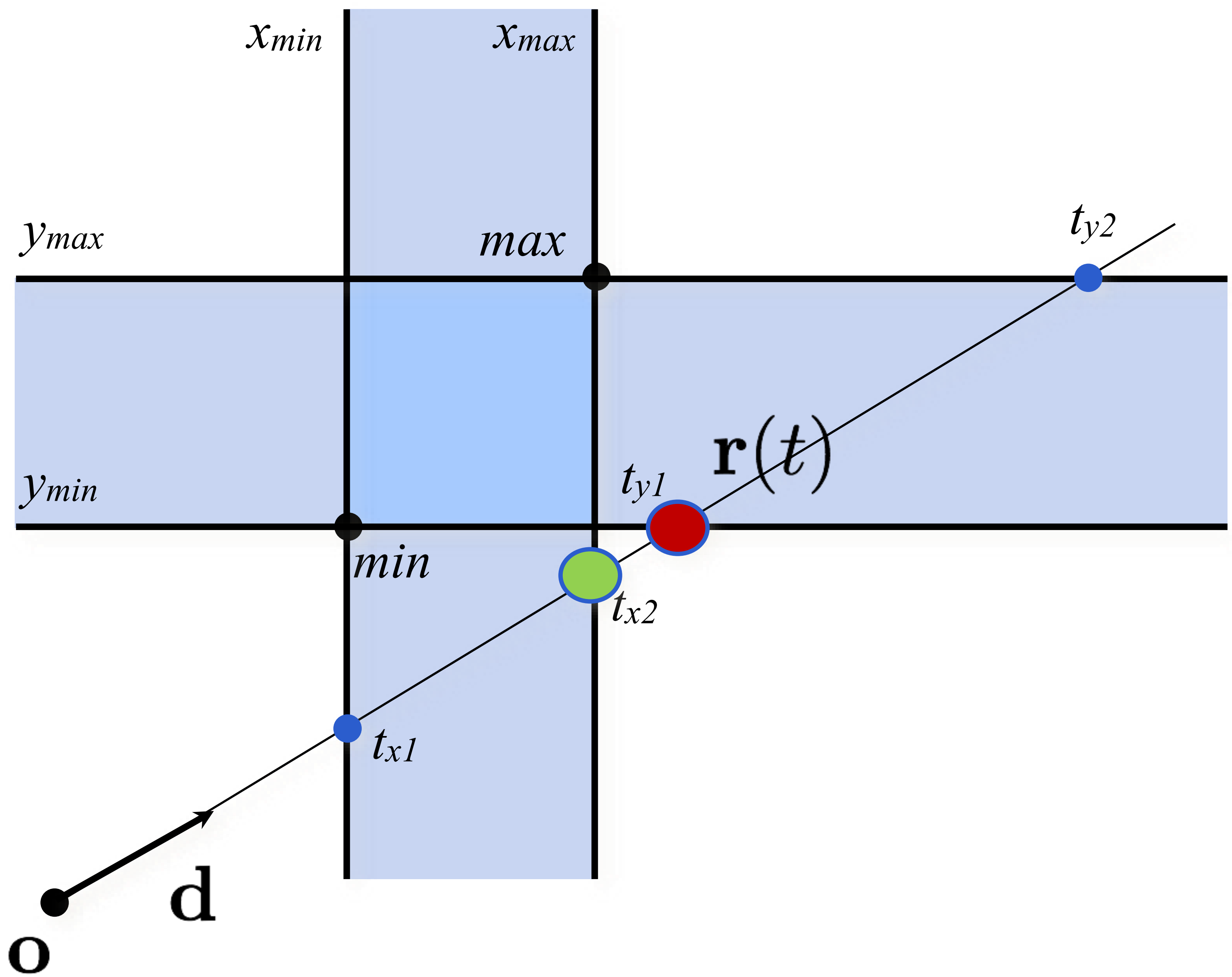
$\boxed{t_{max}} = \min(t_{x2}, t_{y2}, t_{z2})$

hit if: $t_{min} < t_{max}$



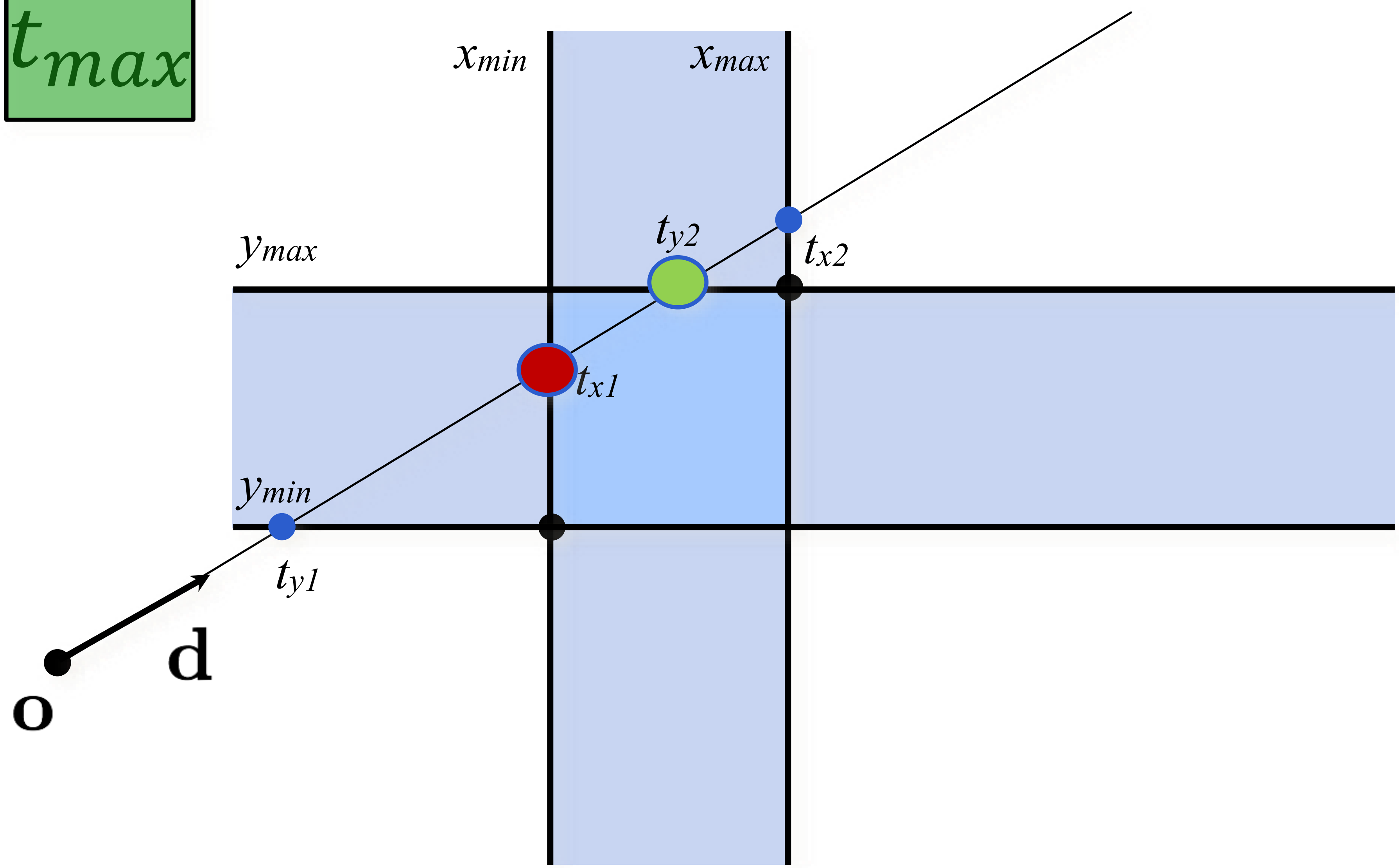
Ray-AABB Intersection

- $\boxed{t_{min}} > \boxed{t_{max}}$



Ray-AABB Intersection

- $\boxed{t_{min}} < \boxed{t_{max}}$



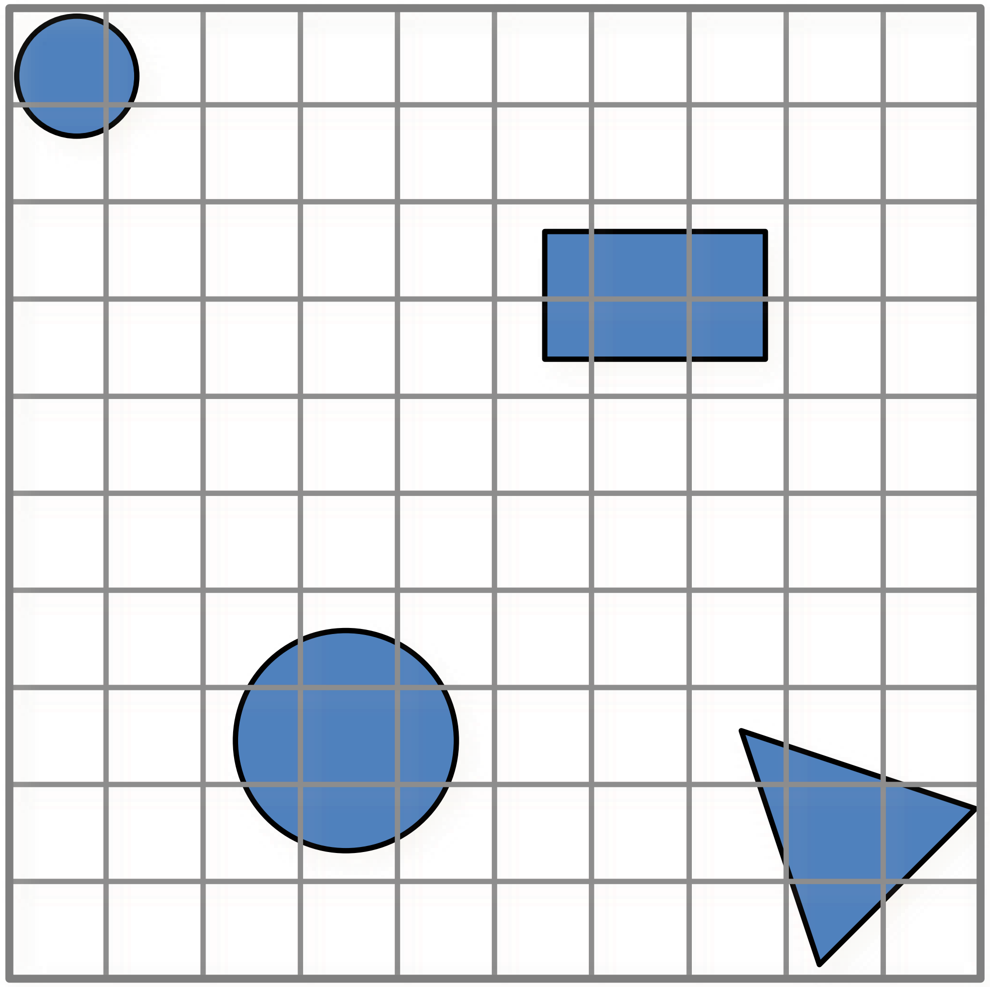
Spatial Sorting

- Preprocess
  - Decompose **space** into disjoint regions
  - Store pointers to overlapping objects within each region

- Rendering
  - Traverse through regions overlapping the ray
  - Intersect objects in each region until a hit is found

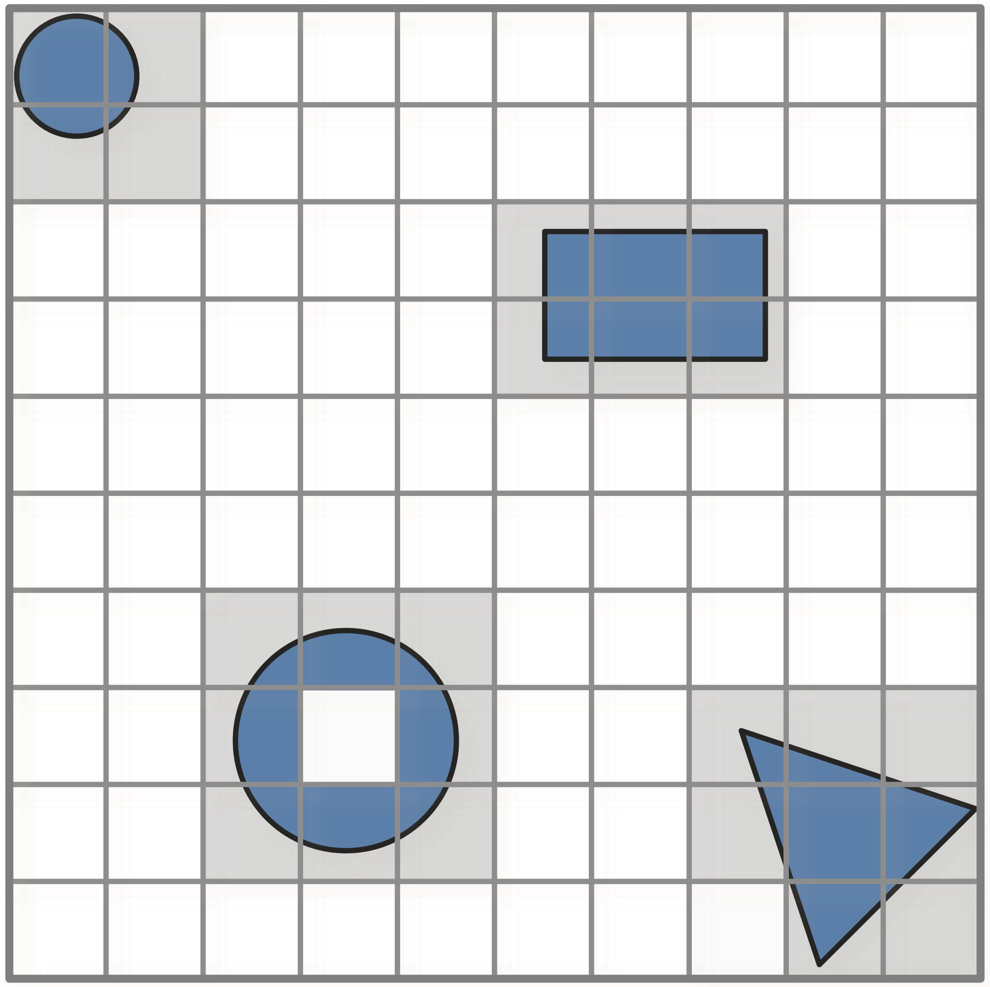
Uniform Grids

- Preprocessing
  - compute bounding box
  - determine grid resolution
    - (often ~ $3\sqrt[3]{n}$)



Uniform Grids

- Preprocessing
  - compute bounding box
  - determine grid resolution
    - (often ~ $3\sqrt[3]{n}$)
  - insert objects into cells
  - Rasterize bounding box
  - Prune empty cells
  - Store reference for each object in cell



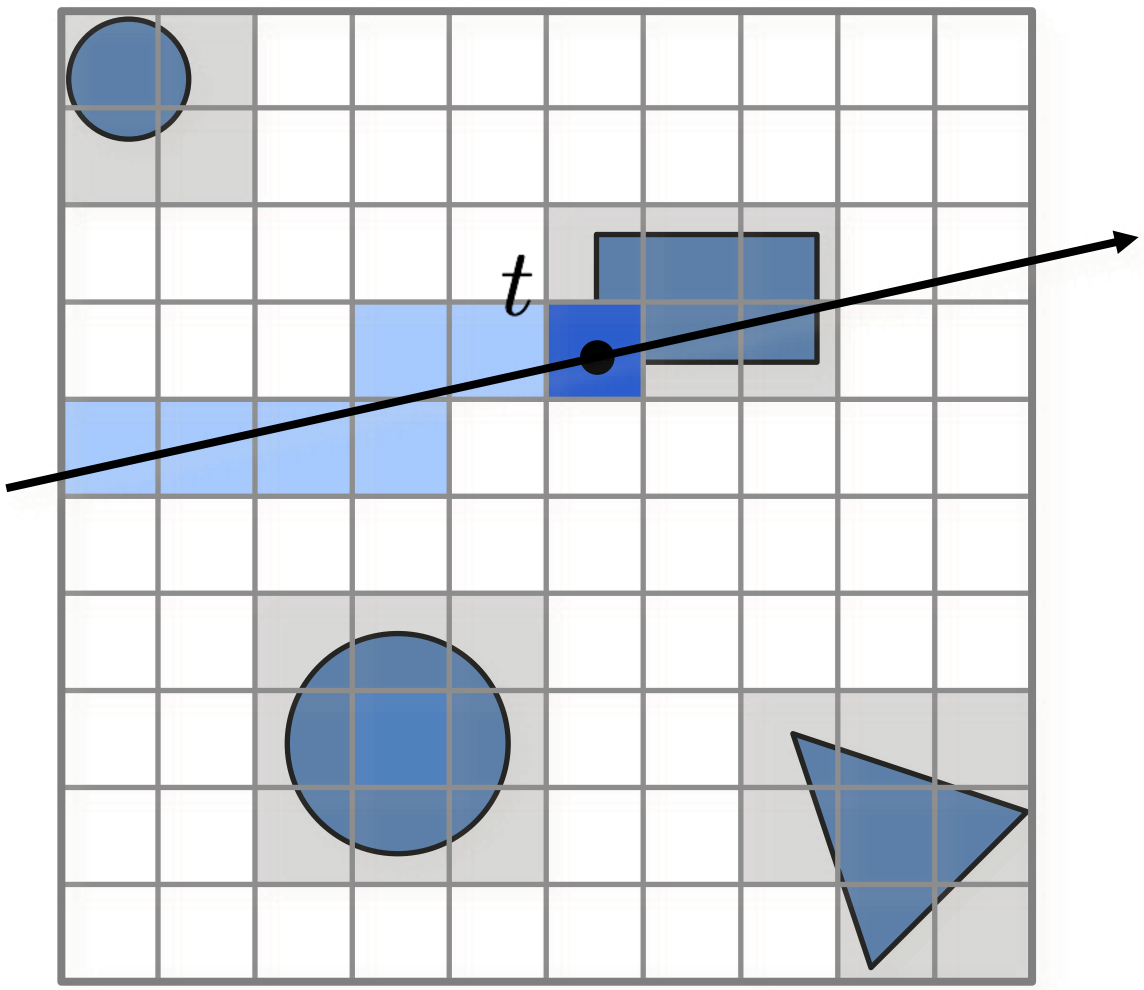
Object/Object Intersections

- In a ray tracer we need to intersect:
  - Rays, planes
  - Spheres, cylinders, cones
  - Triangles/Polygons
  - Axis aligned & oriented bounding boxes
  - etc.

- Implementation reference
  - http://www.realtimerendering.com/intersections.html

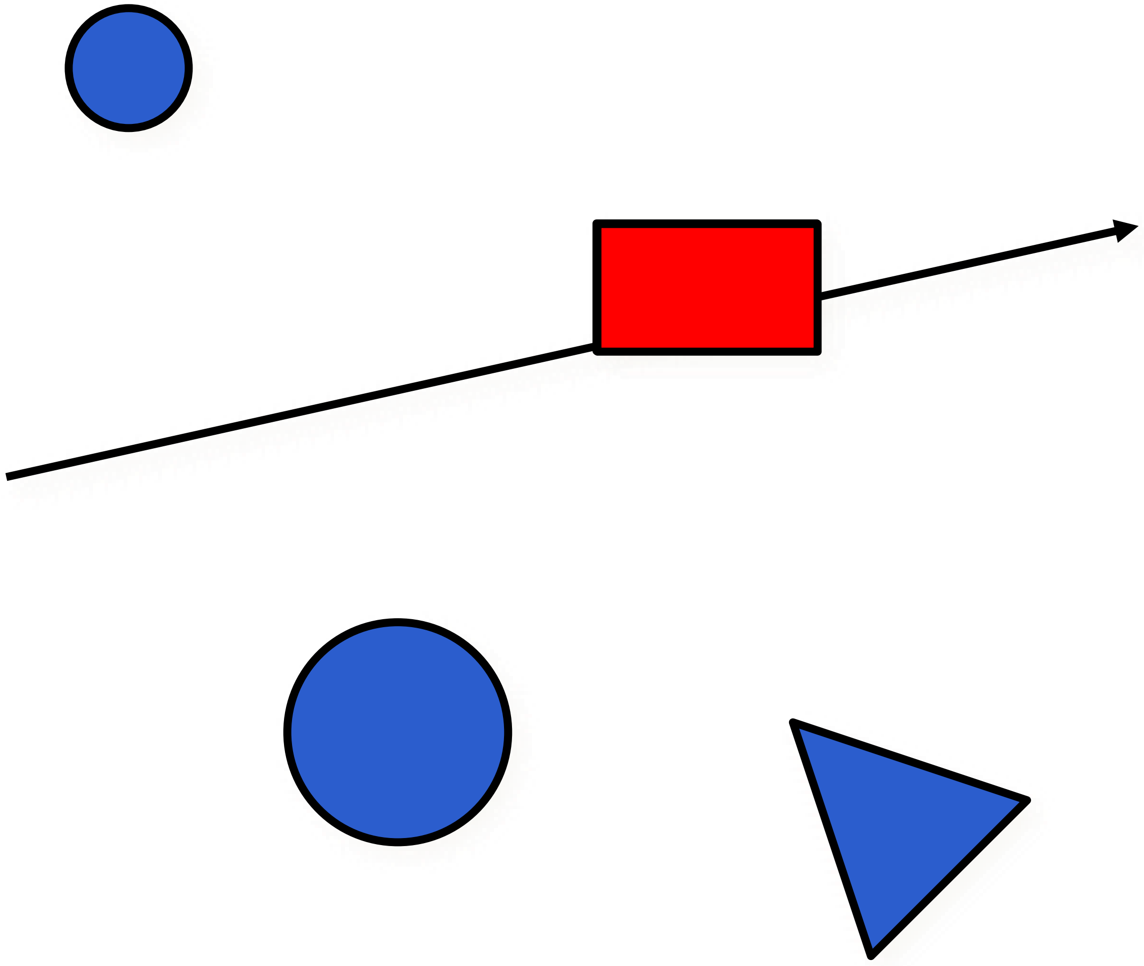
Uniform Grids

- Traversal
  - incrementally rasterize ray
  - compute intersection with objects in each cell
  - stop when intersection found in current voxel

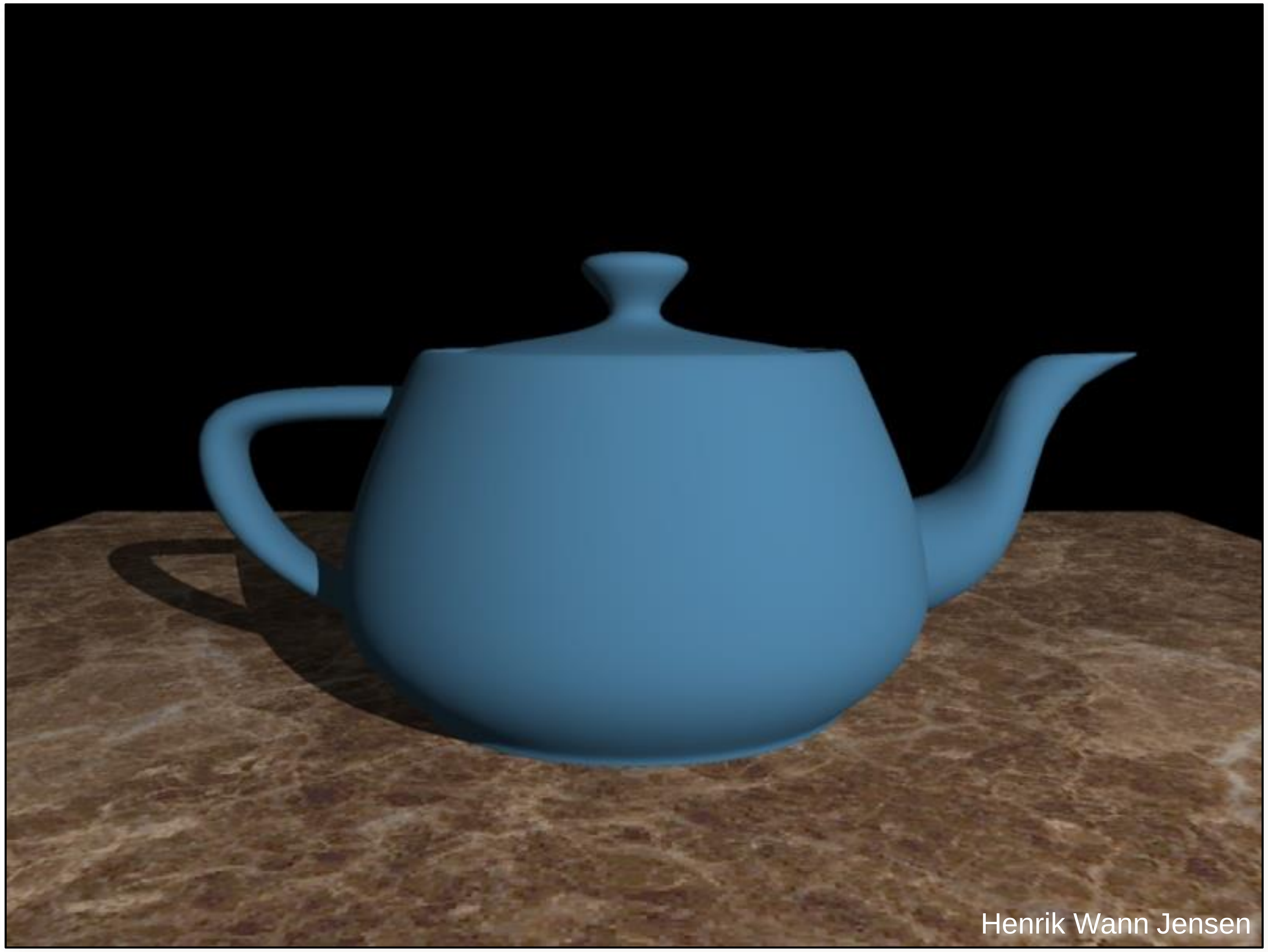


Uniform Grids

- Comparison: brute-force
  - intersect ray with every primitive
  - take closest intersection



Uniform Grid Efficiency

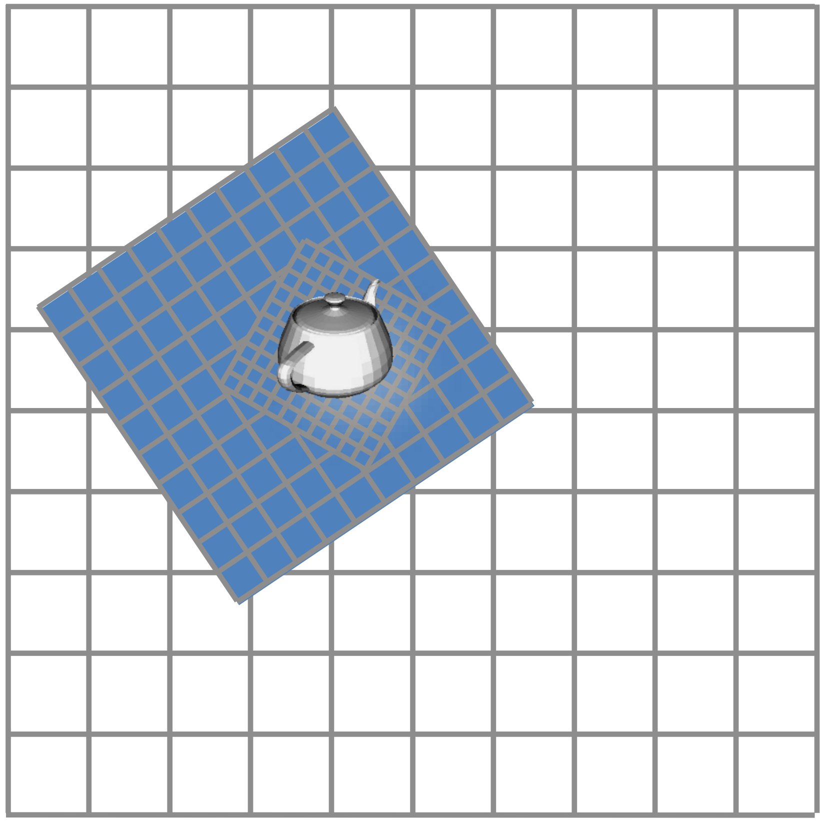


6321 triangles

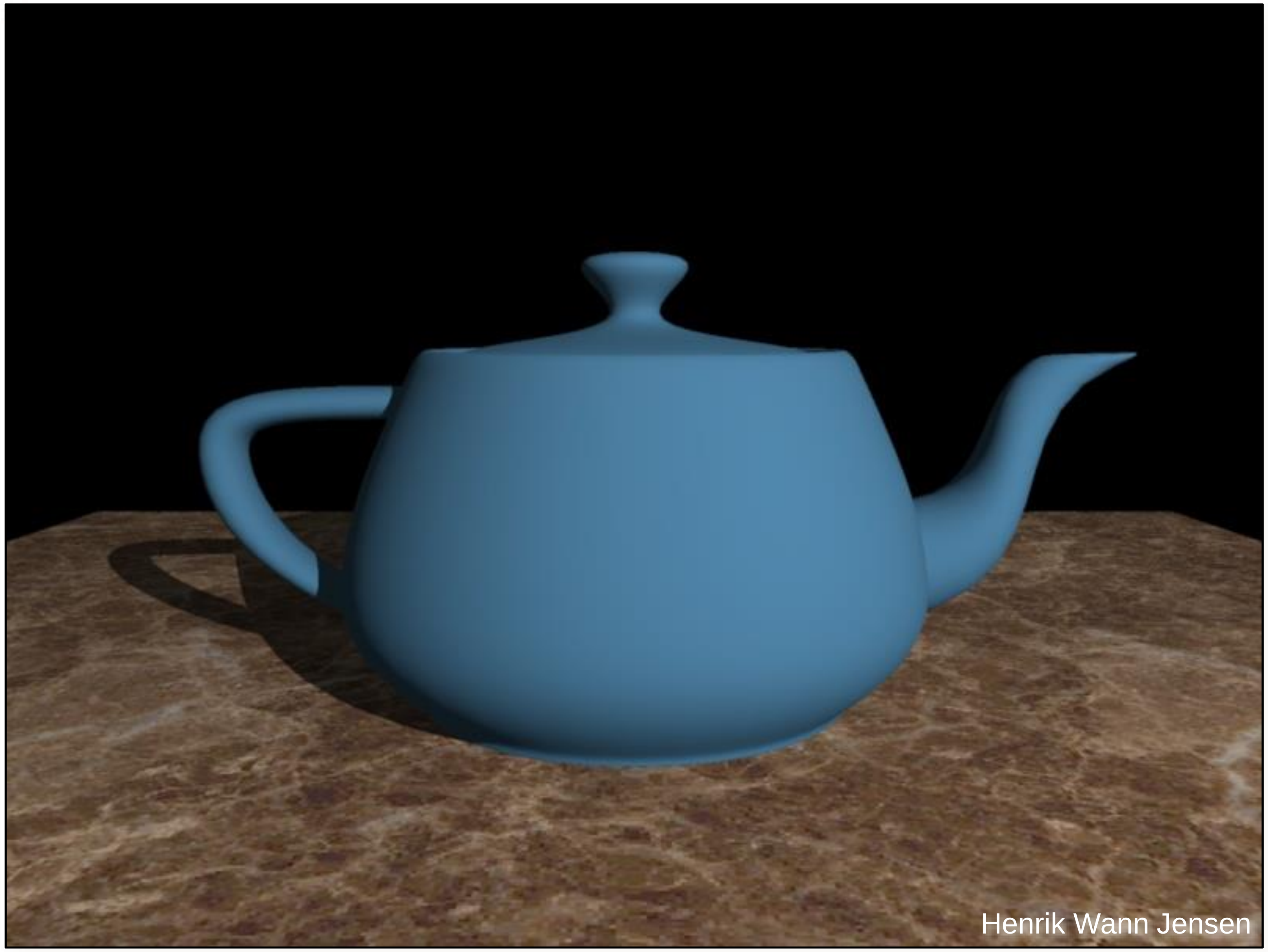
- Brute force: 6321 intersection tests per ray (total = 3,710,882,127)
- Uniform grid: 44.86 intersection tests per ray (total = 26,336,575)

Uniform Grids

- Advantages
  - Easy to code, building data structure is fast
- Disadvantages
  - Uniform cells do not adapt to non-uniform scenes
    - Teapot in a stadium problem
  - Hierarchical grids



Hierarchical Grid Efficiency



- Brute force: 6321 intersection tests per ray (total = 3,710,882,127)
- Uniform grid: 44.86 intersection tests per ray (total = 26,336,575)
- 2-level grid: 12.05 intersection tests per ray (total = 7,072,774)

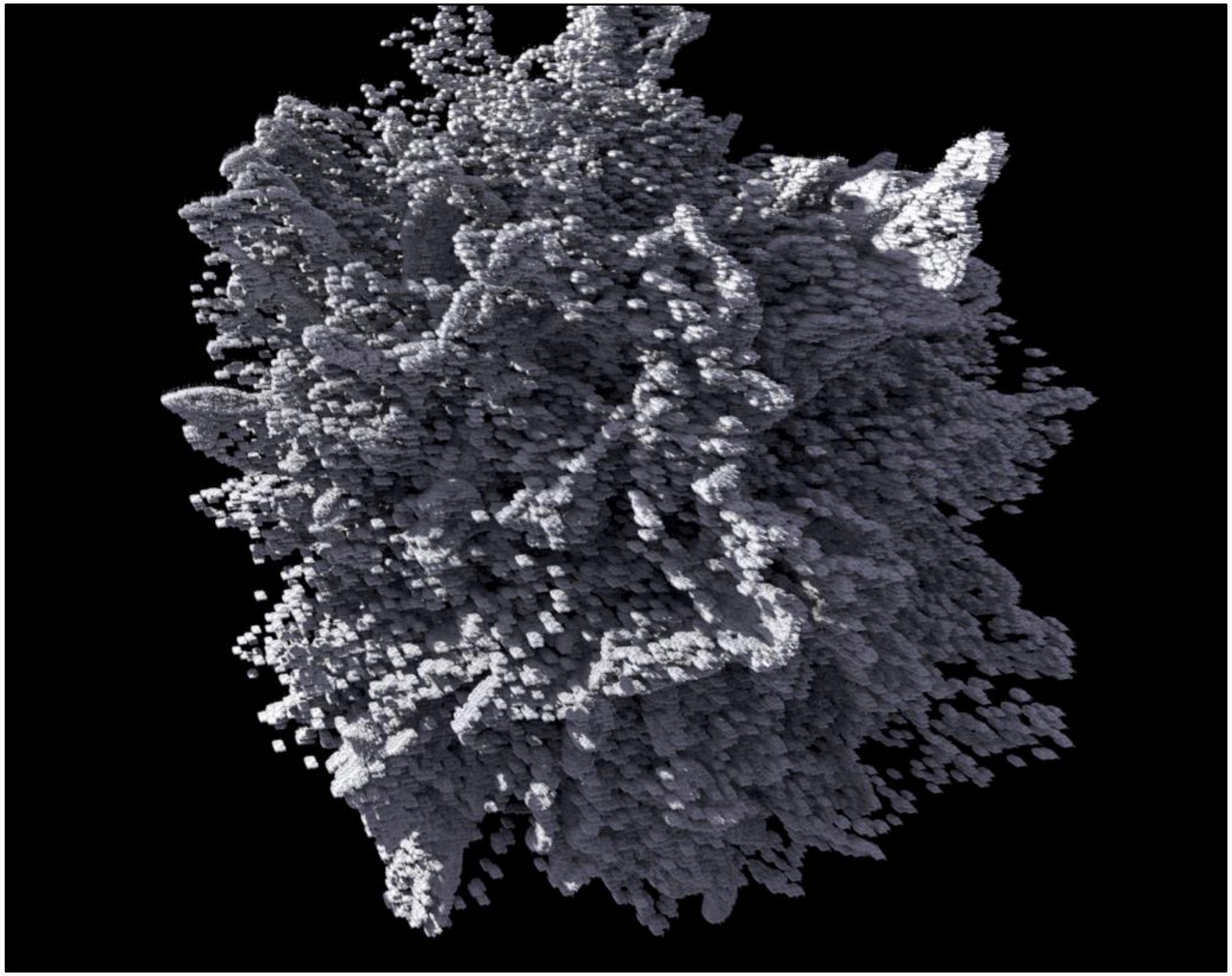
Complex Geometry

- Grass

render time: 7 minutes

Visual Break

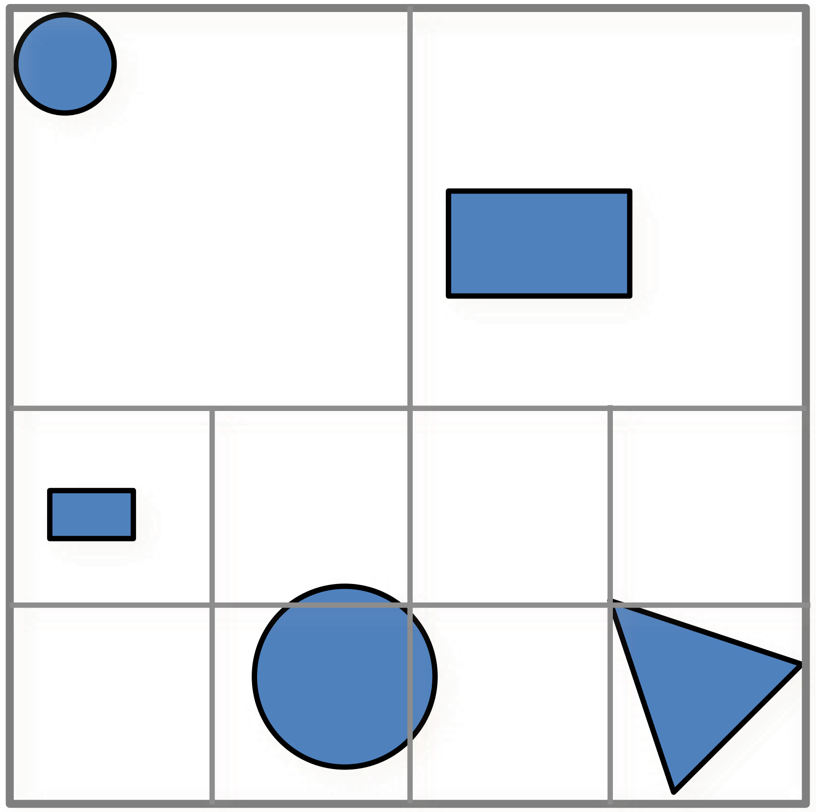
A Complex Scene



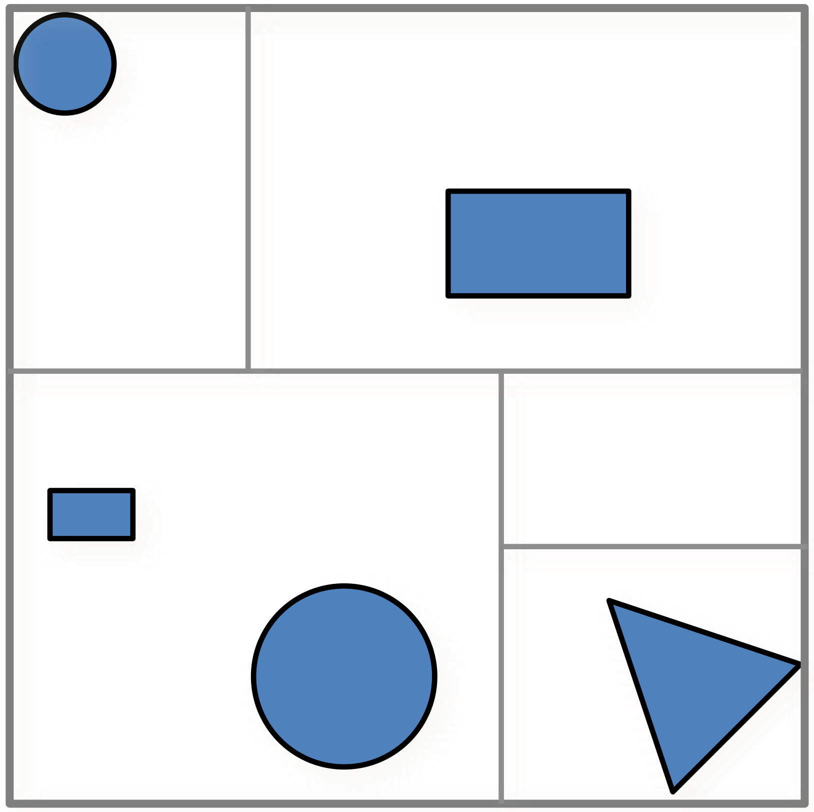
245 billion polys. 250,000 instances.

Spatial Hierarchies

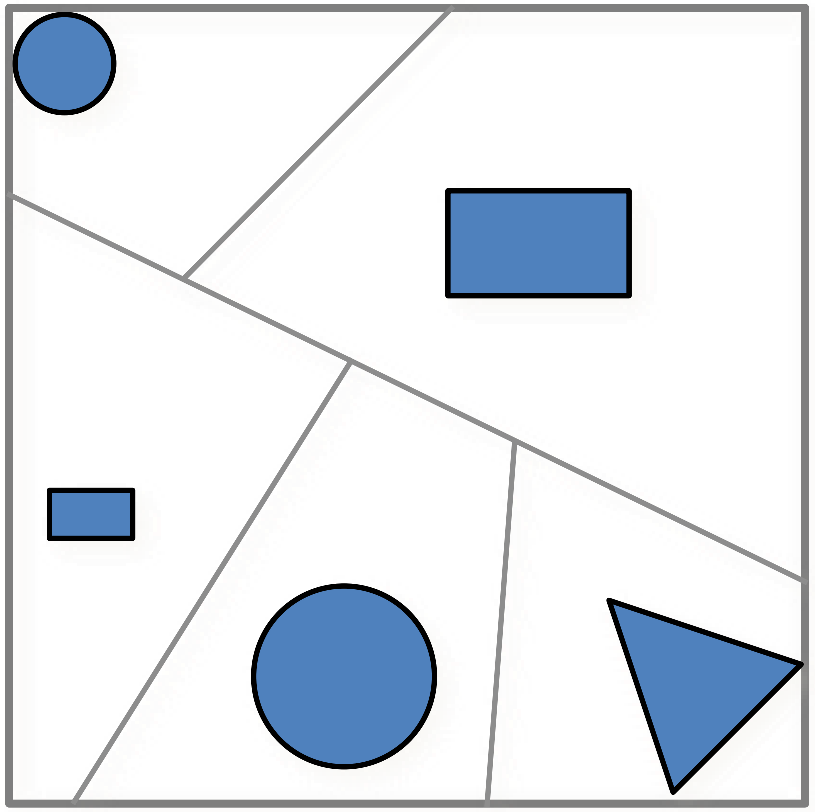
- Classical divide-and-conquer approach
- Several variations



octree



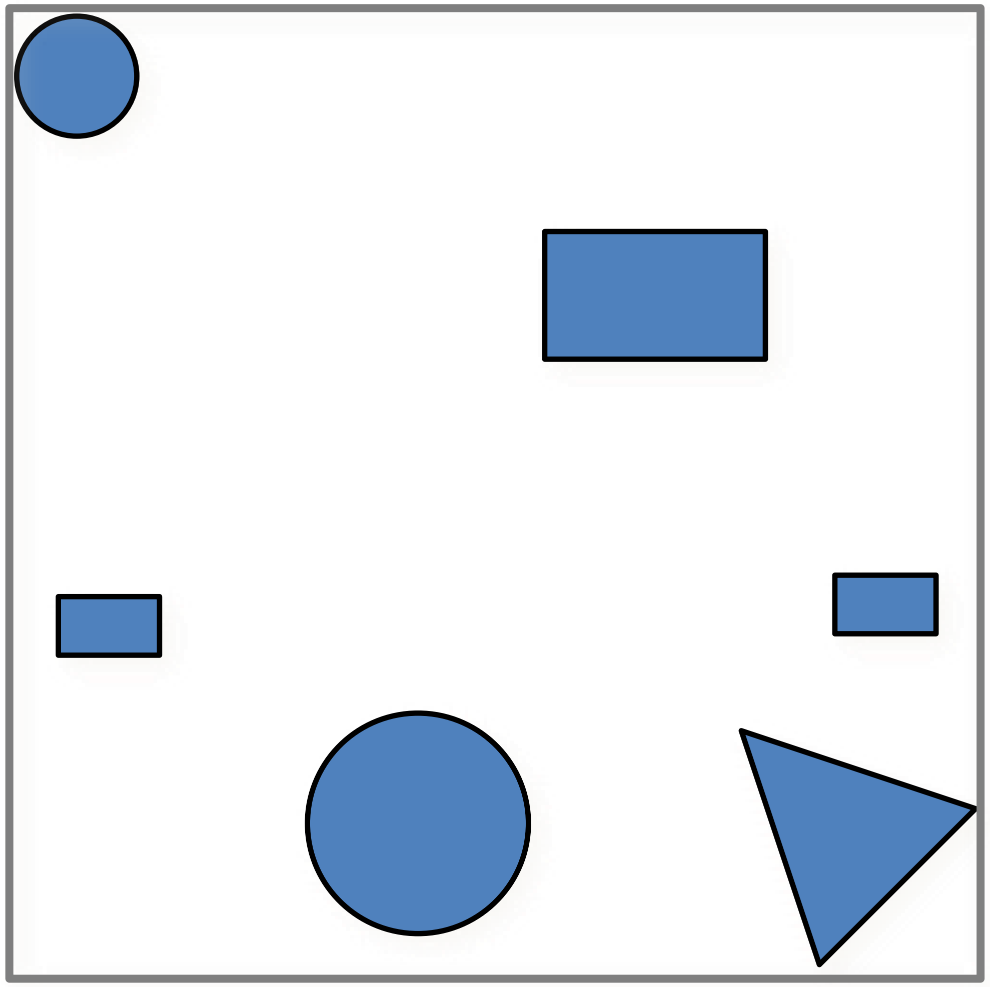
kd-tree



bsp-tree

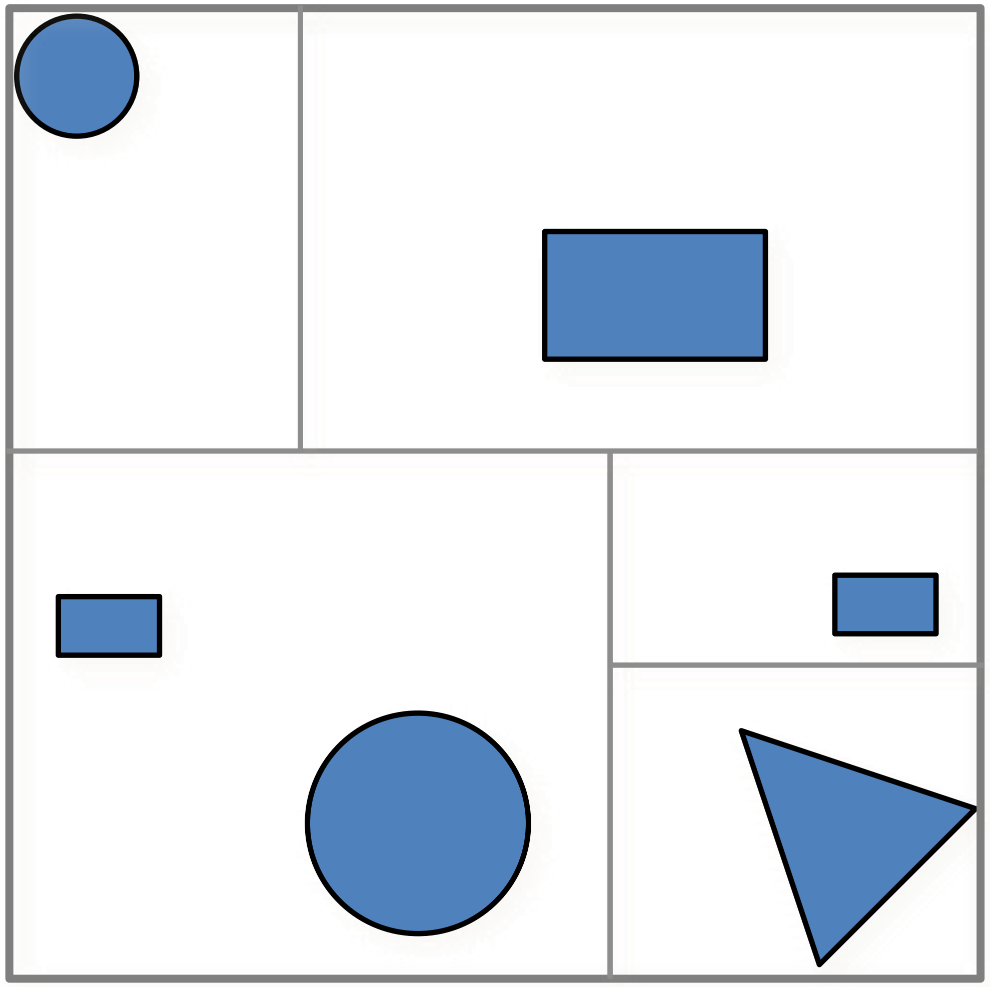
KD-Trees

- Preprocessing
  - compute bounding box



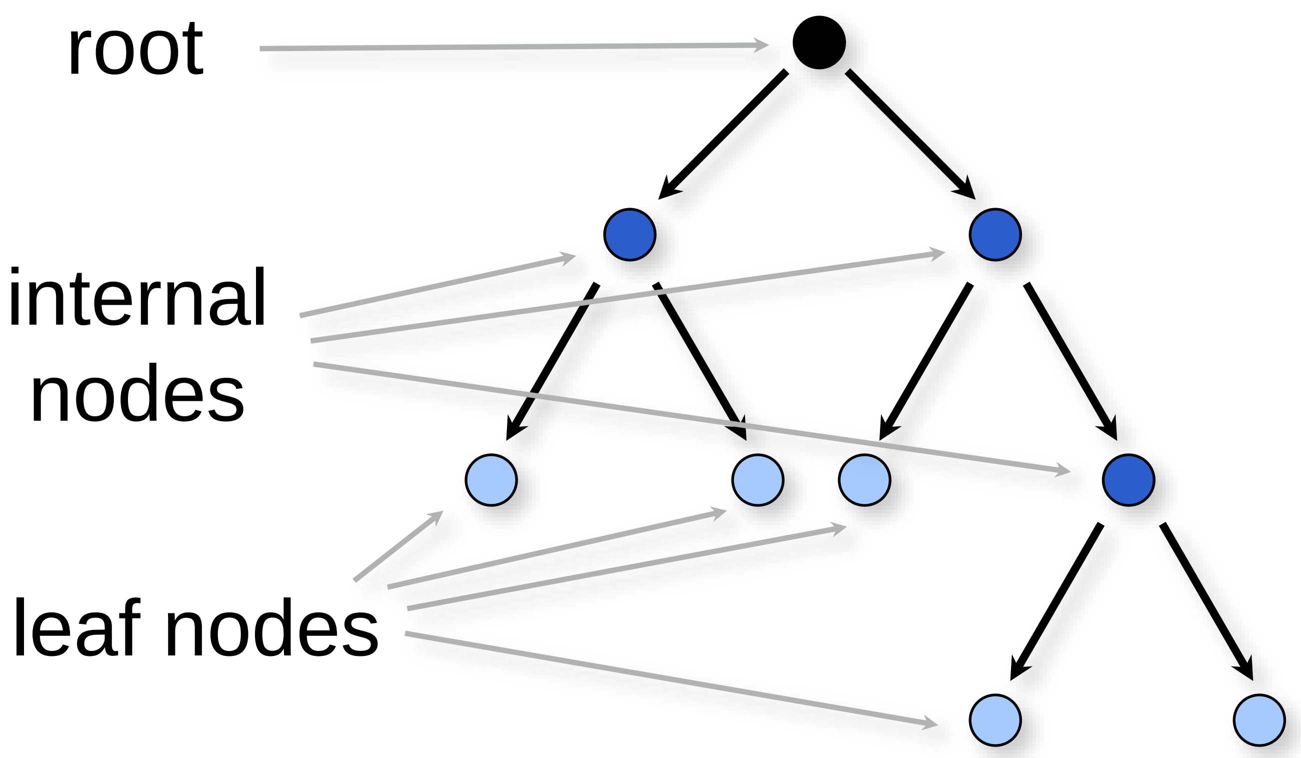
KD-Trees

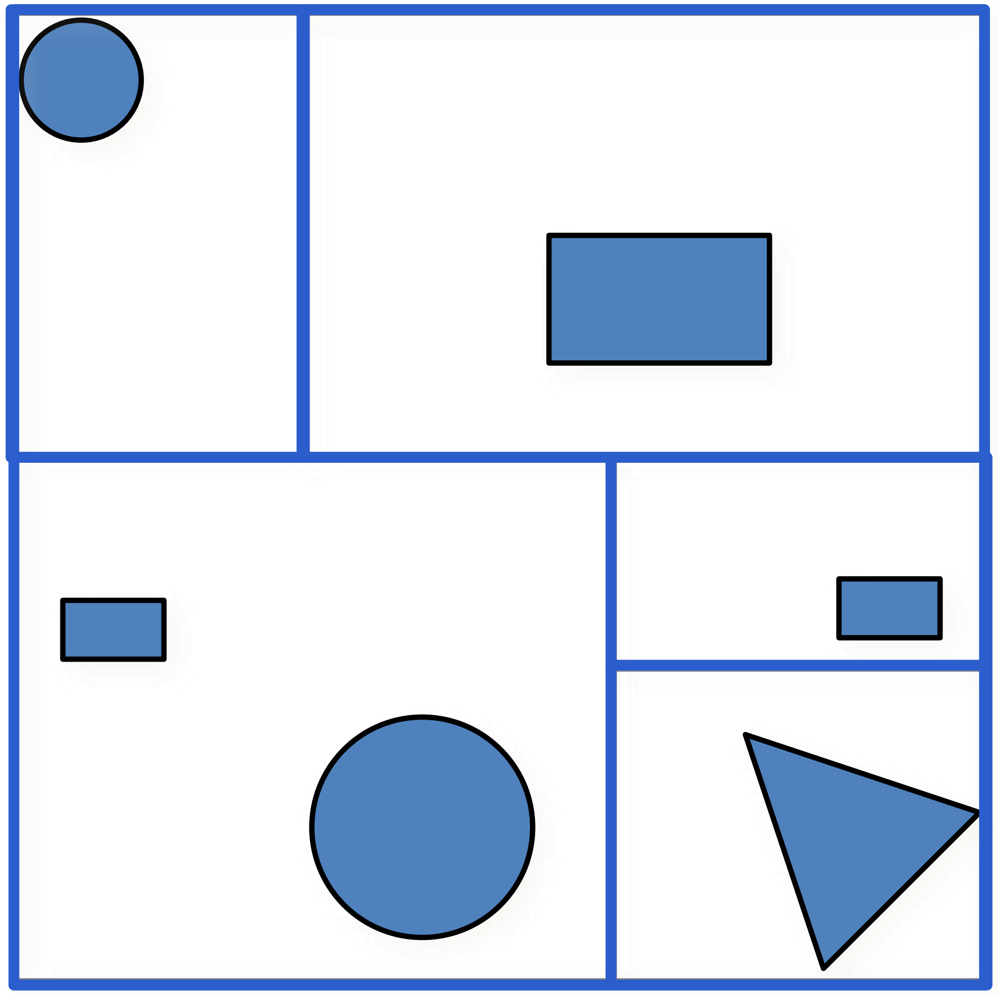
- Preprocessing
  - compute bounding box
  - recursively split cell using axis-aligned plane
  - until termination criteria e.g. maximum depth or minimum number of objects



KD-Trees

- Preprocessing
  - binary tree structure





only leaf nodes store reference to geometry!

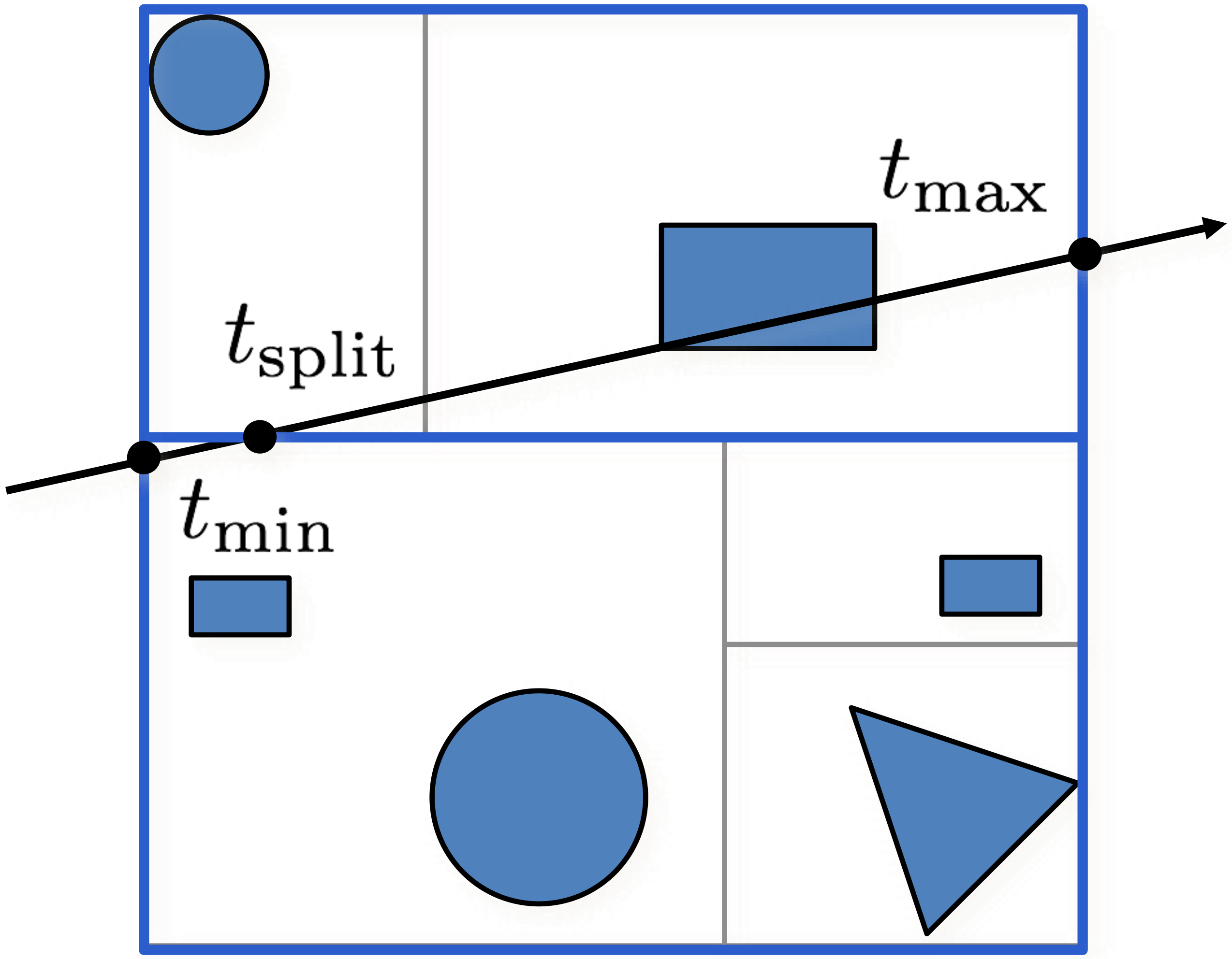
KD-Trees

- Internal nodes store
  - split axis: x-, y-, or z-axis
  - split position: coordinate of split plane along axis
  - children: reference to child nodes
- Leaf nodes store
  - list of primitives
  - optionally: mailboxing information

KD-Trees

- Traversal
  - top-down recursion

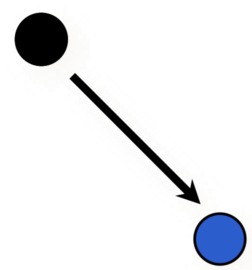
•

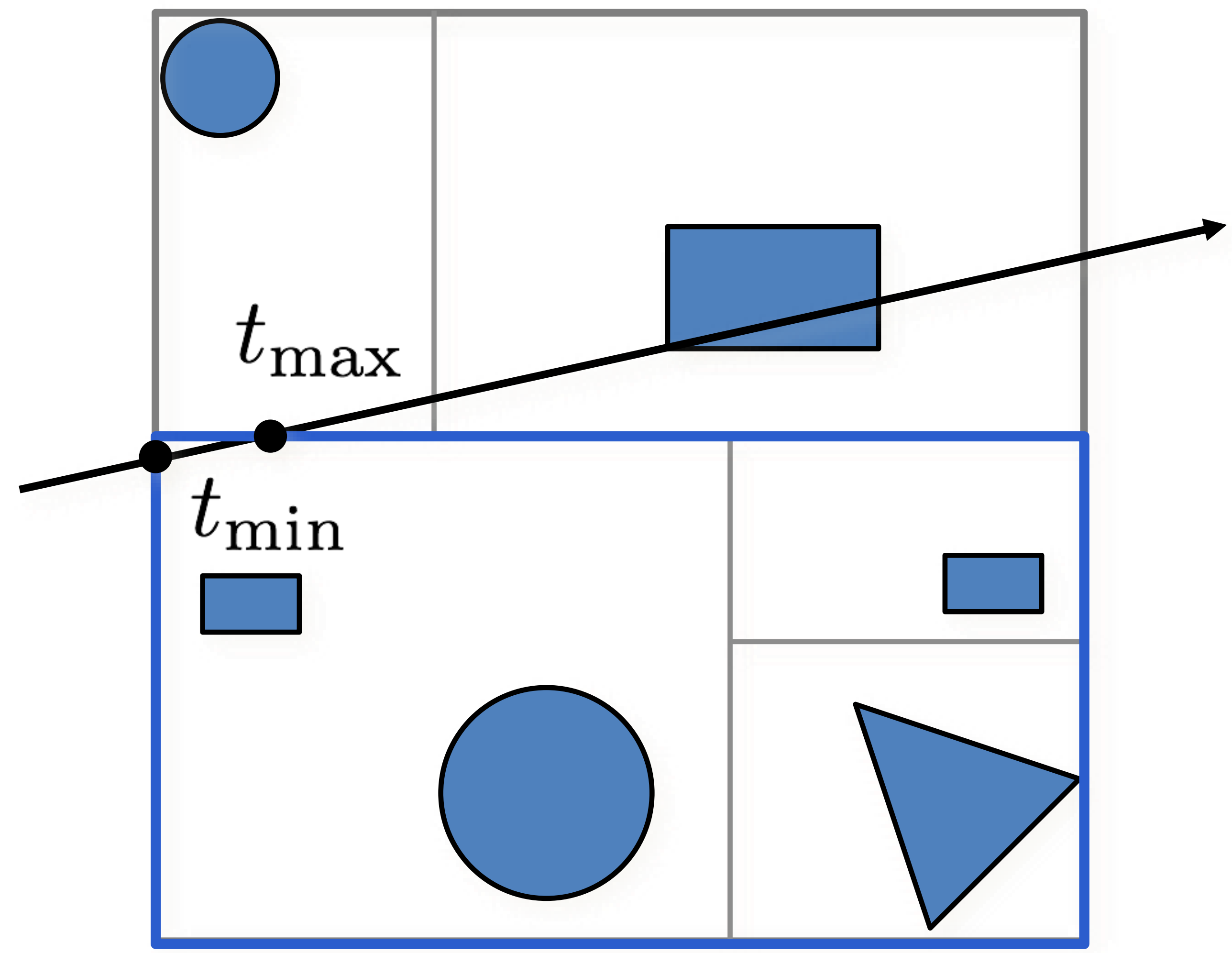


internal node → split

KD-Trees

- Traversal
  - top-down recursion

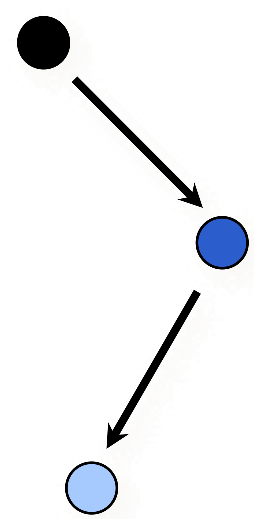


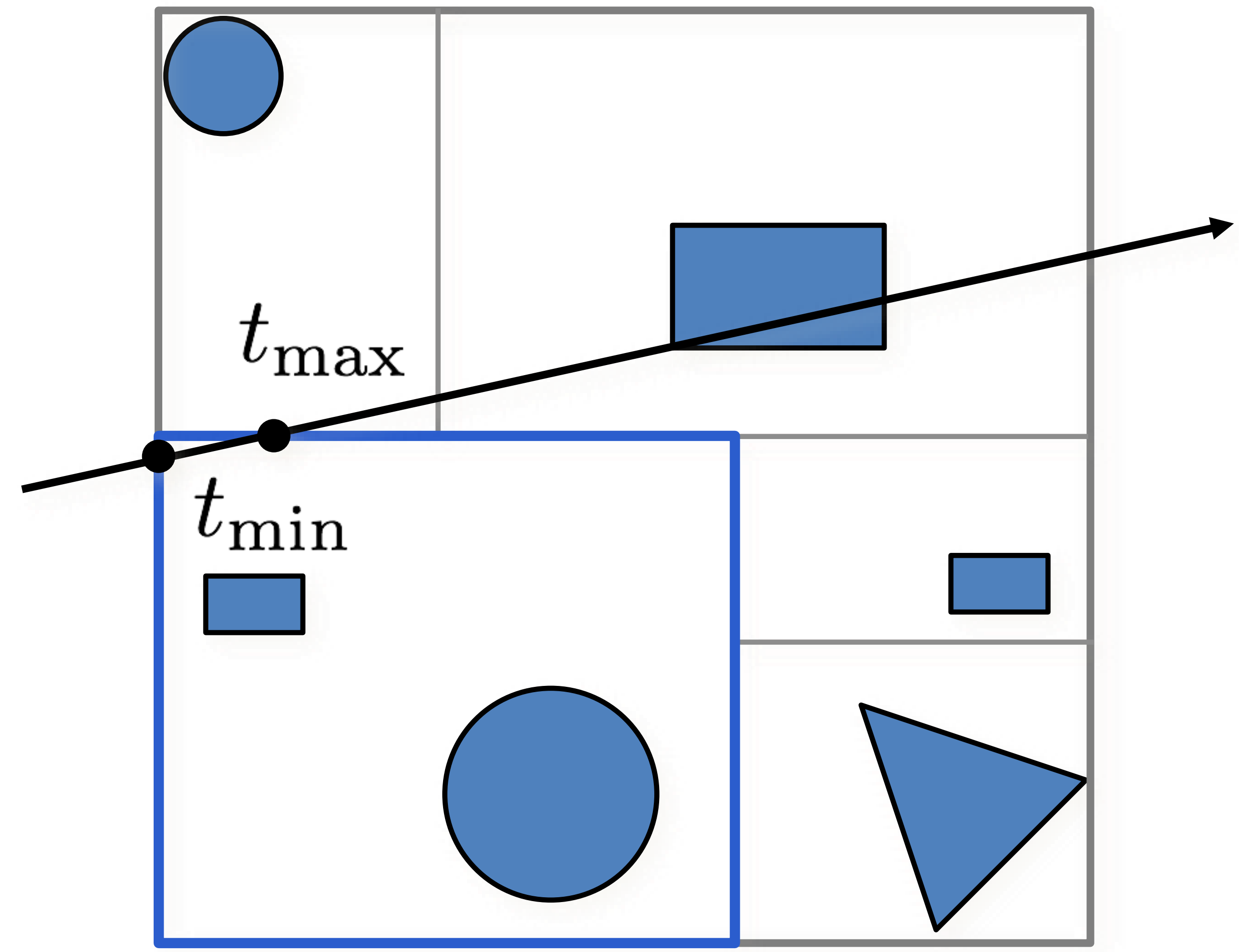


internal node → split

KD-Trees

- Traversal
  - top-down recursion

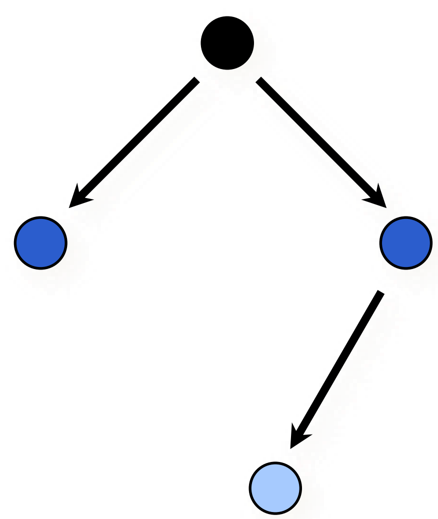


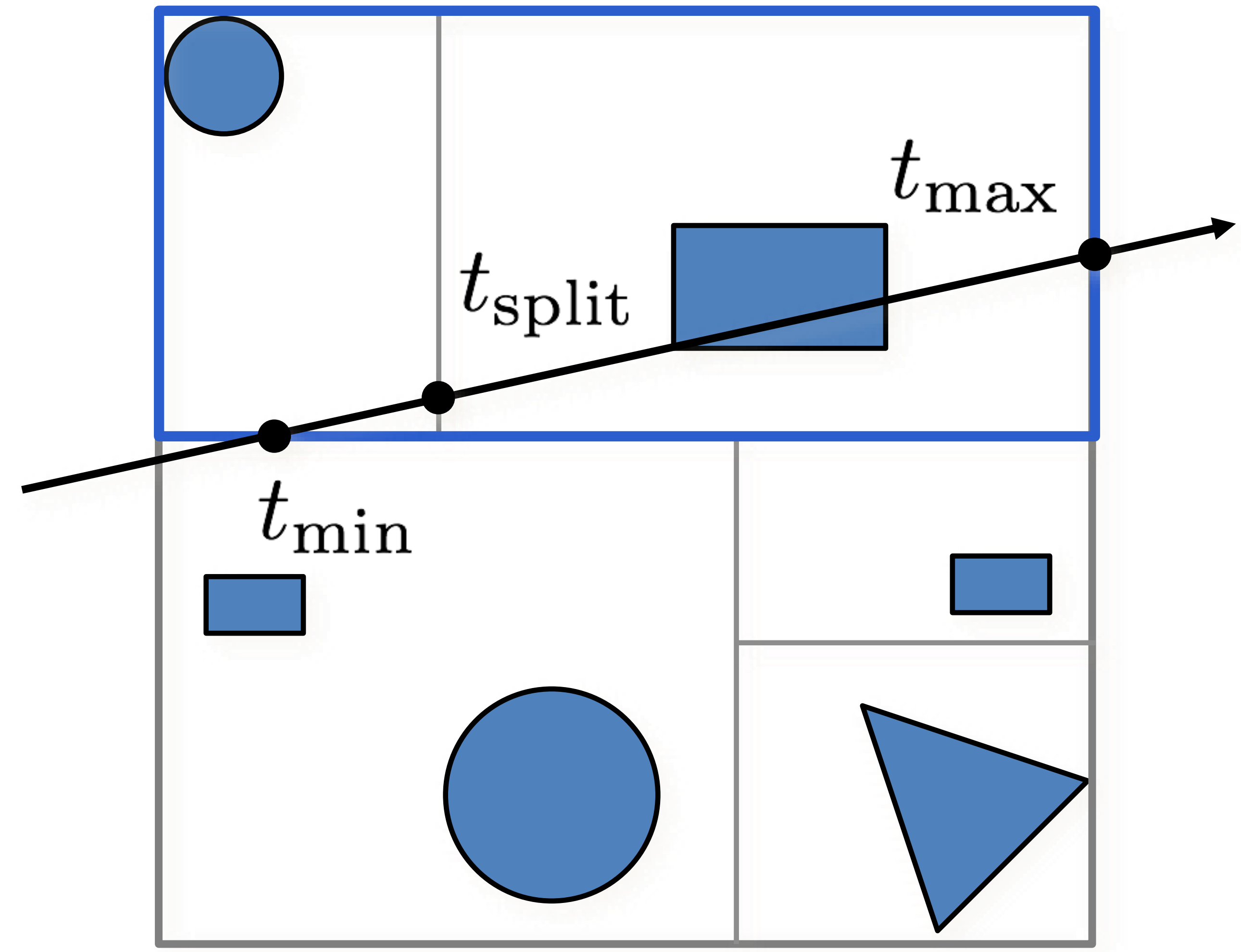


leaf node → intersect

KD-Trees

- Traversal
  - top-down recursion

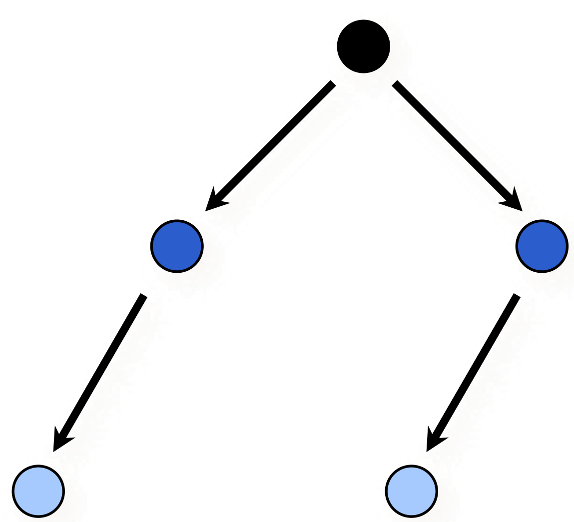


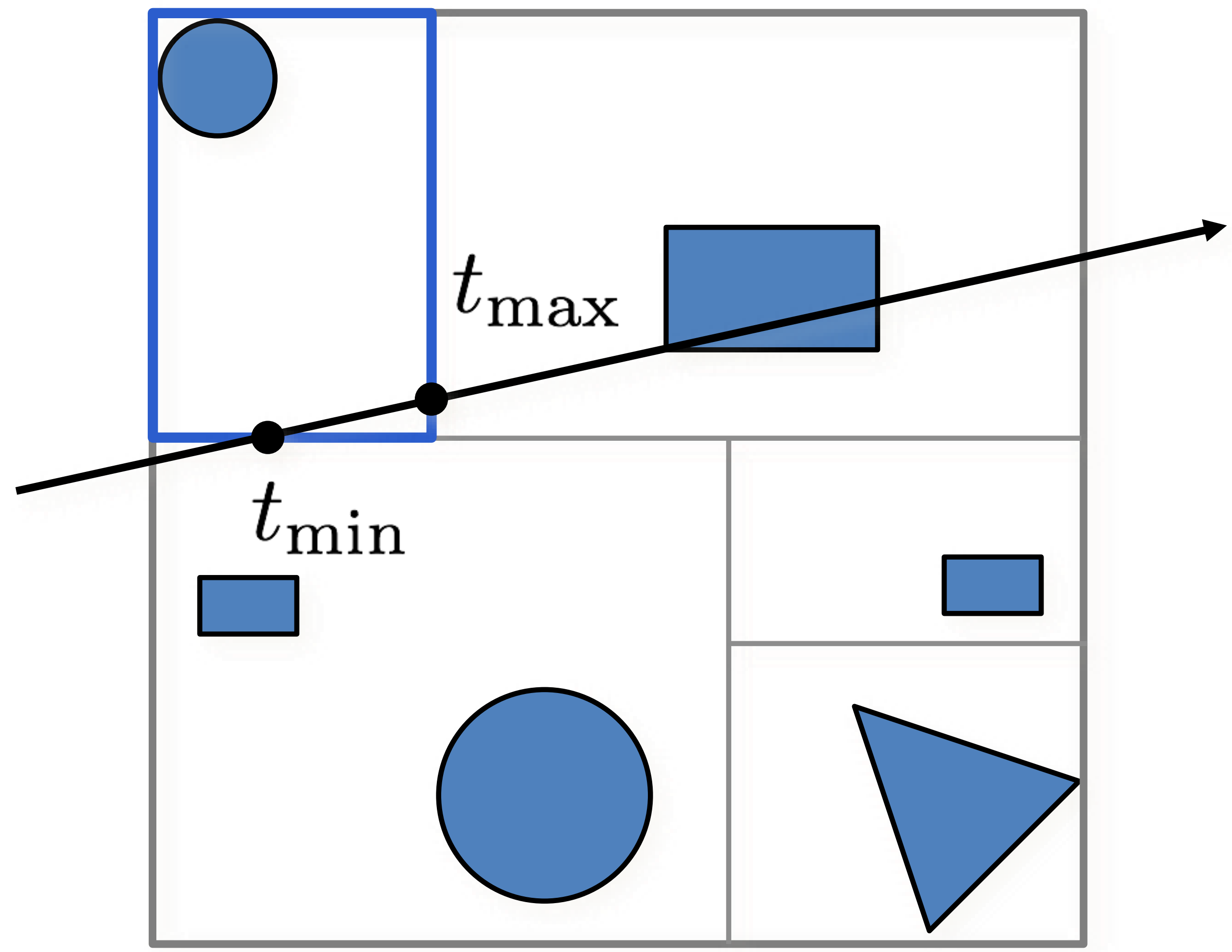


internal node → split

KD-Trees

- Traversal
  - top-down recursion

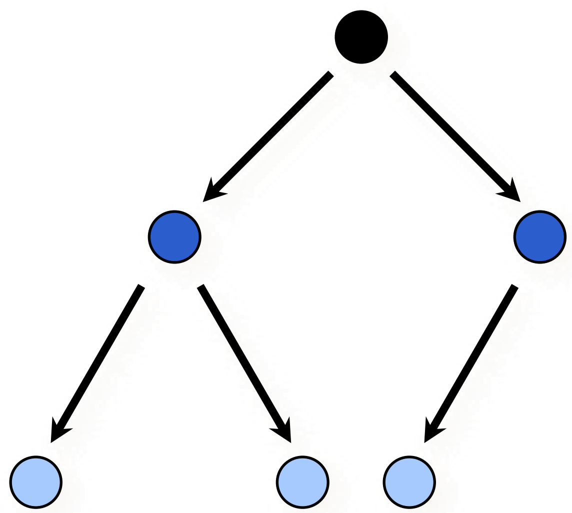


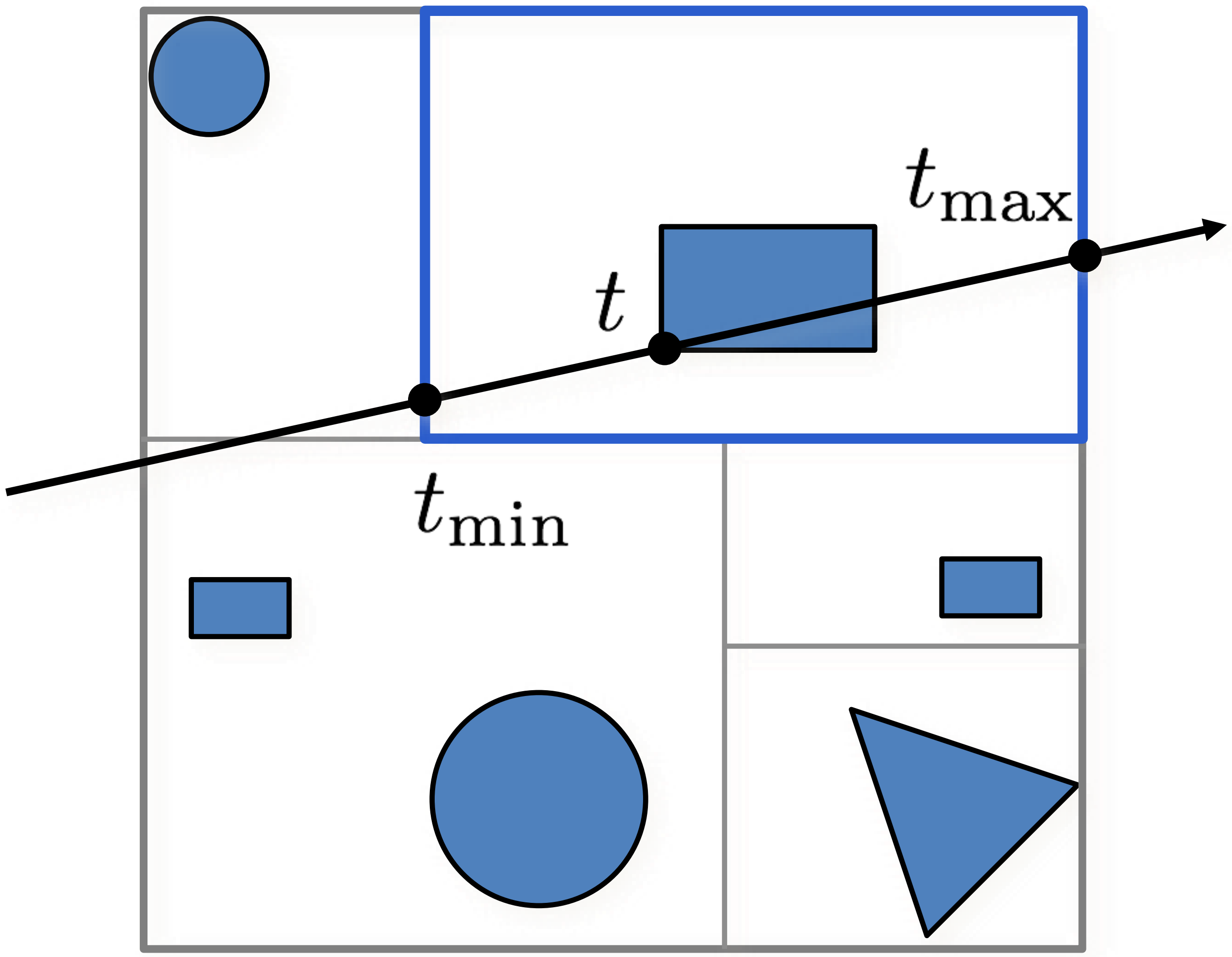


leaf node → intersect

KD-Trees

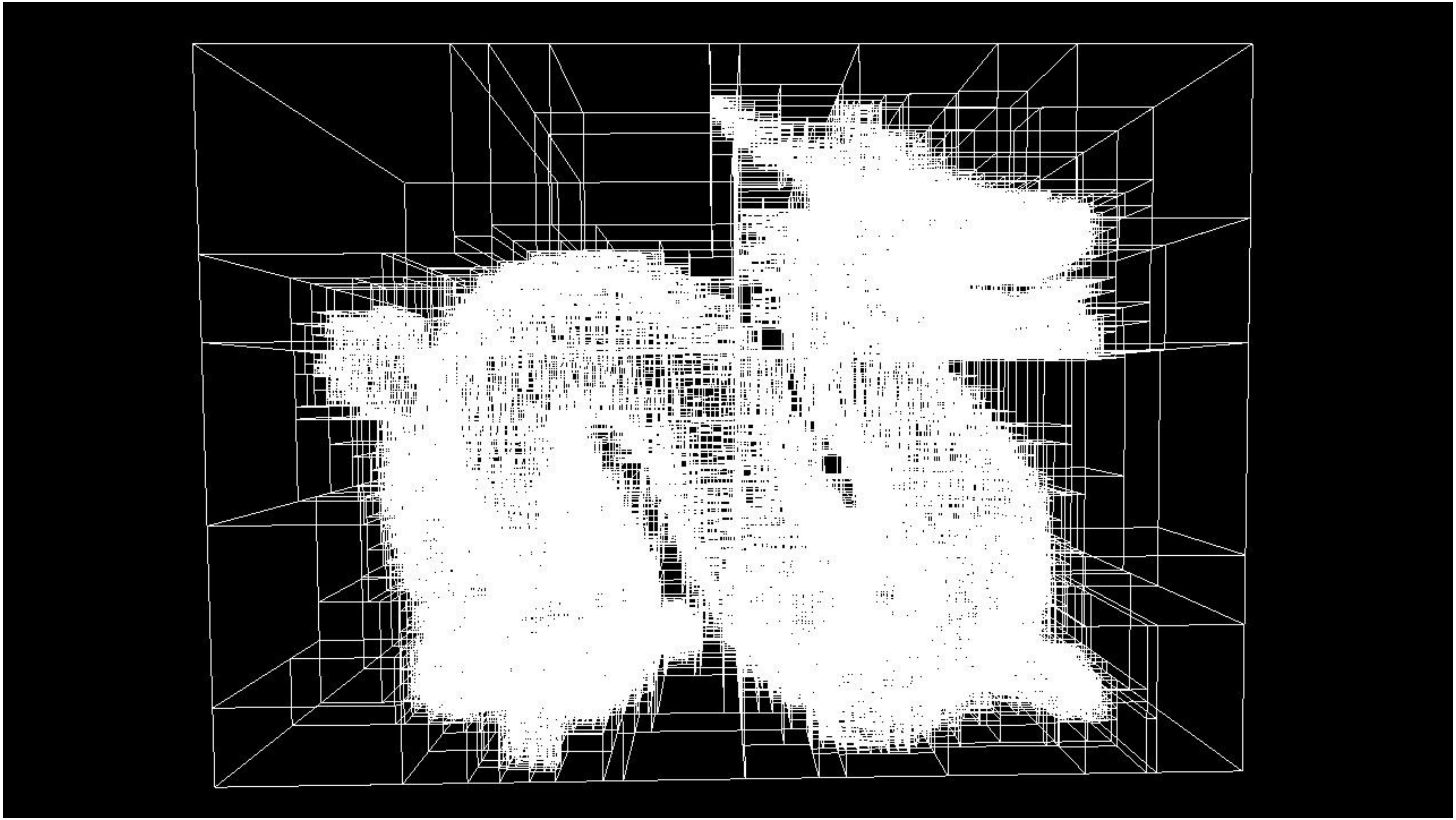
- Traversal
  - top-down recursion



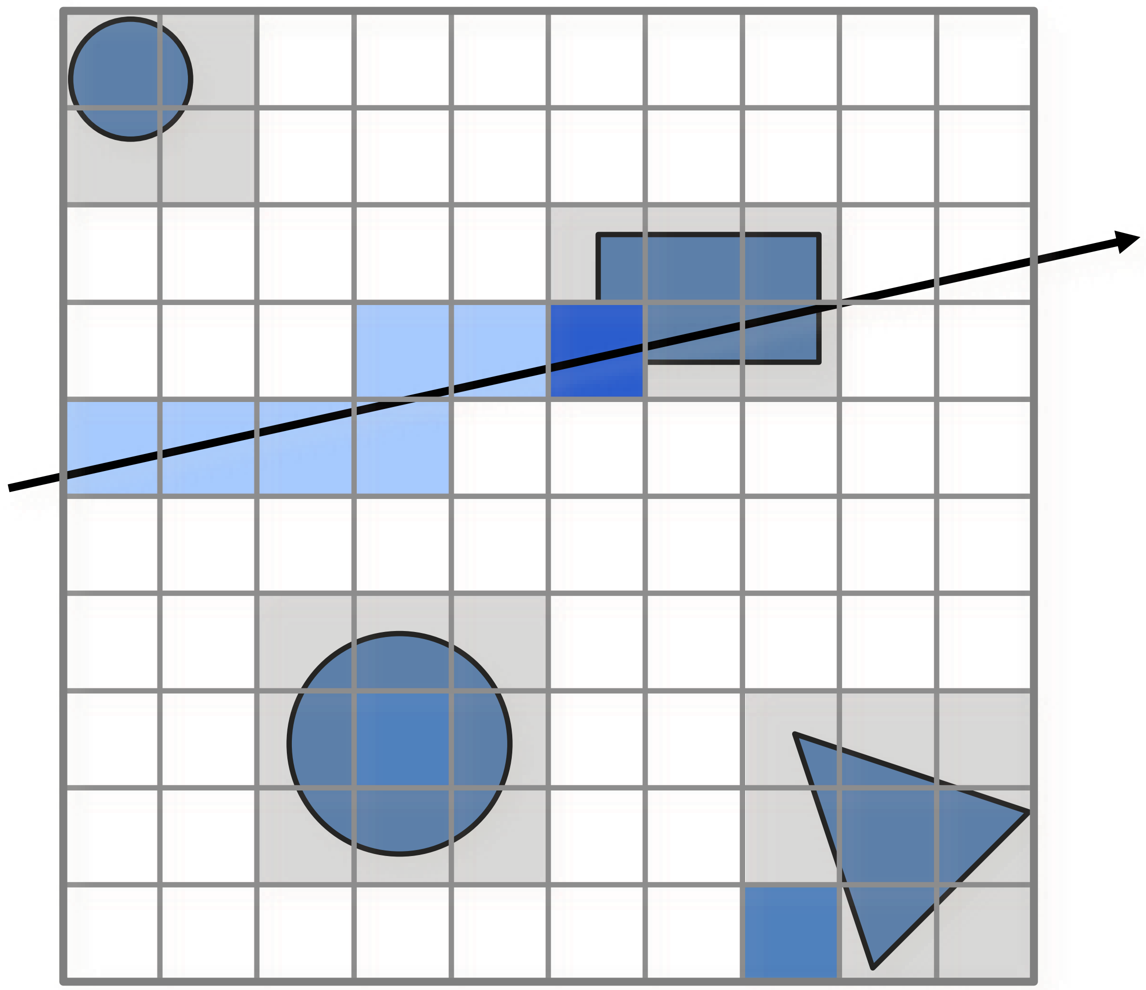


leaf node → intersect

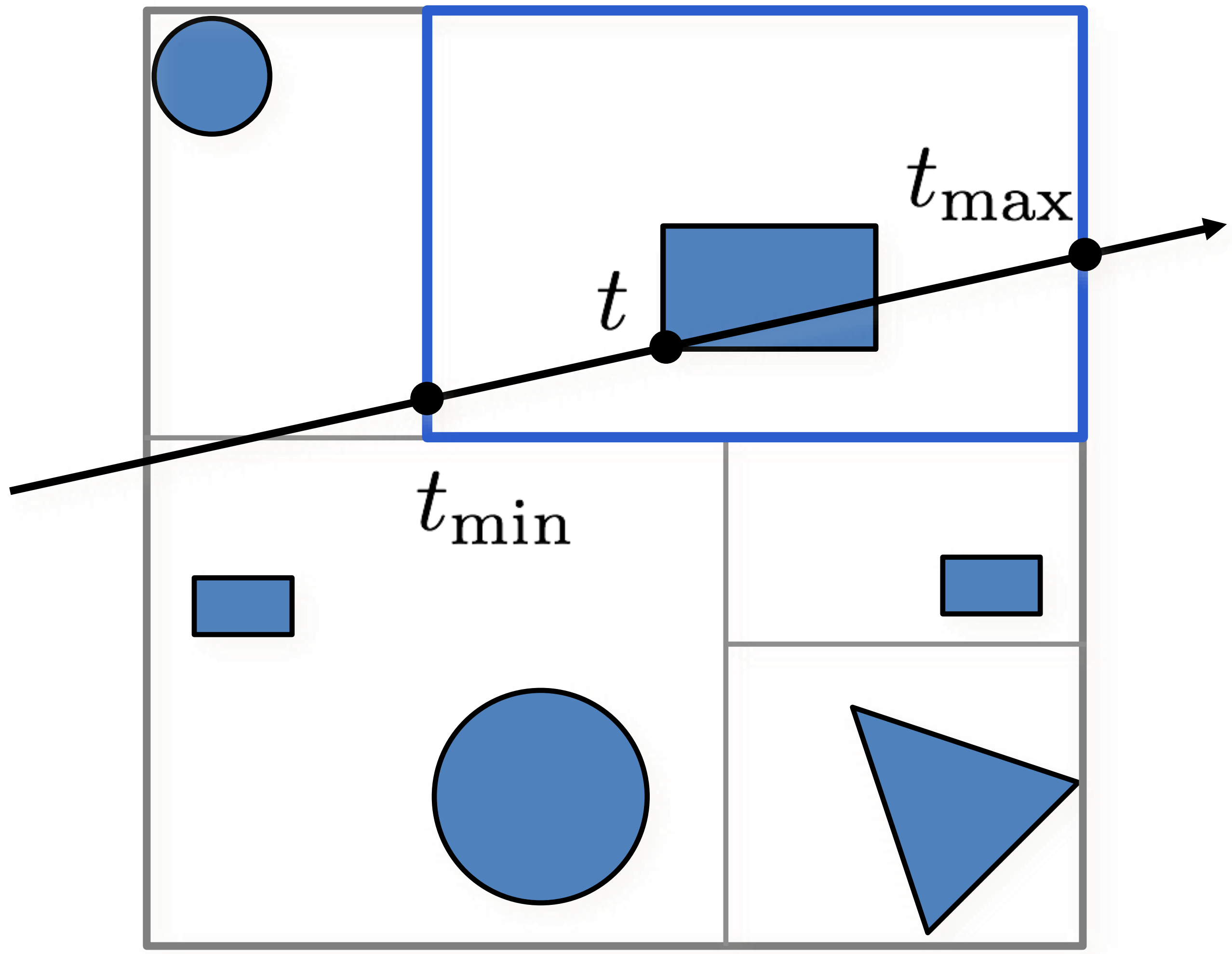
KD-Tree



Comparison

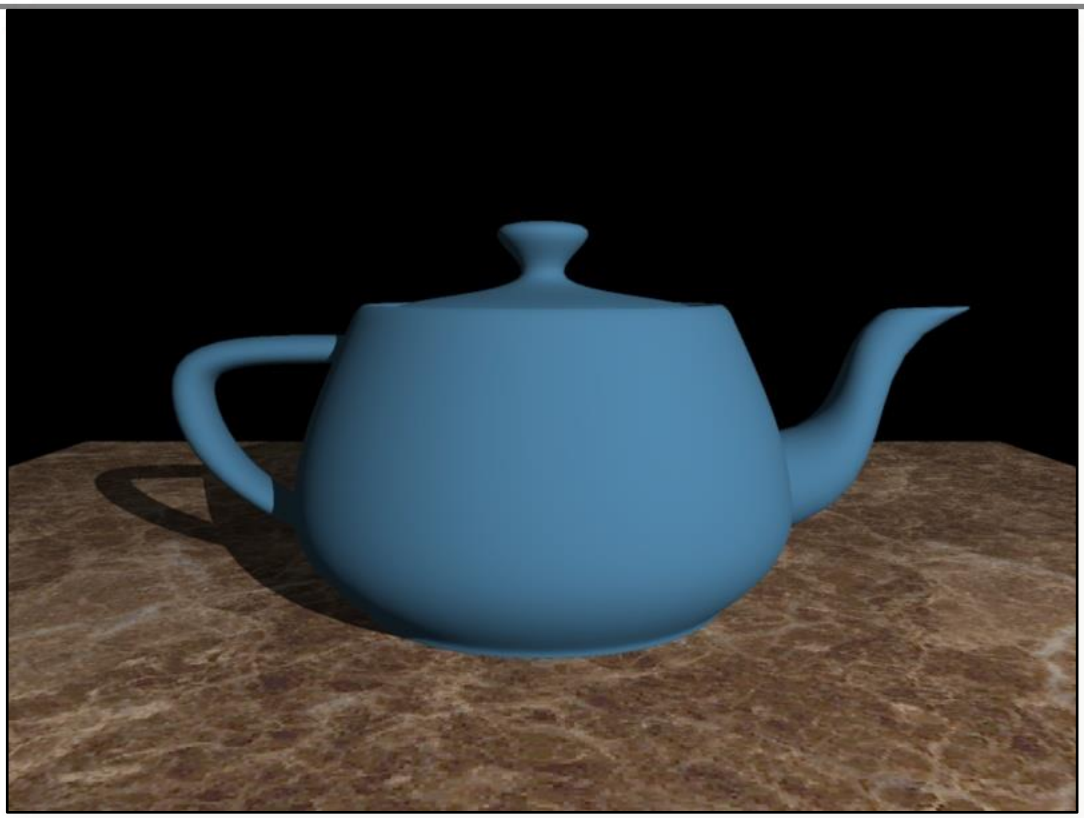


uniform grid



kd-tree

KD-Tree Efficiency

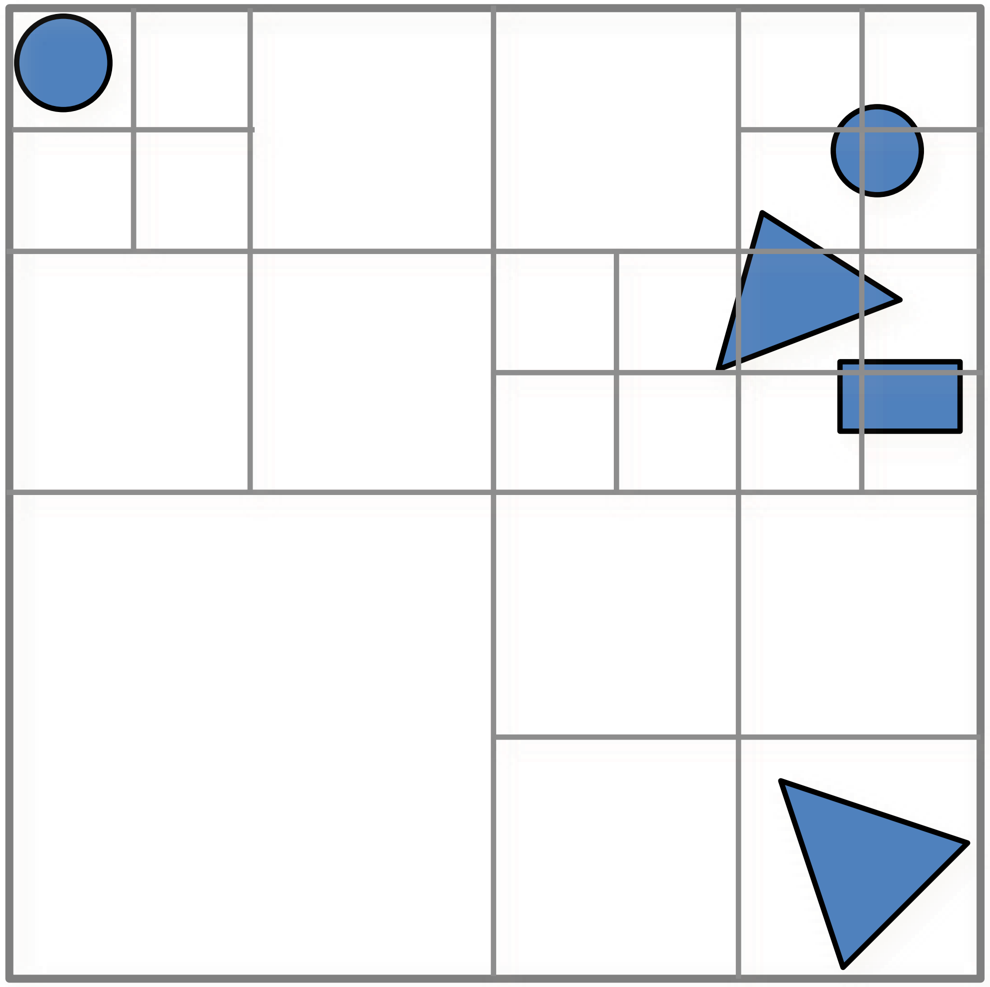


	Total intersection tests	Intersection tests / ray
Brute force	9,986,402,697	6321.00
depth=8, mo=10	111,204,795	70.38
depth=16, mo=8	11,361,140	7.19
depth=24, mo=8	9,930,604	6.28
depth=24, mo=4	6,350,655	4.02
depth=32, mo=2	4,426,580	2.80

Visual Break

Octrees

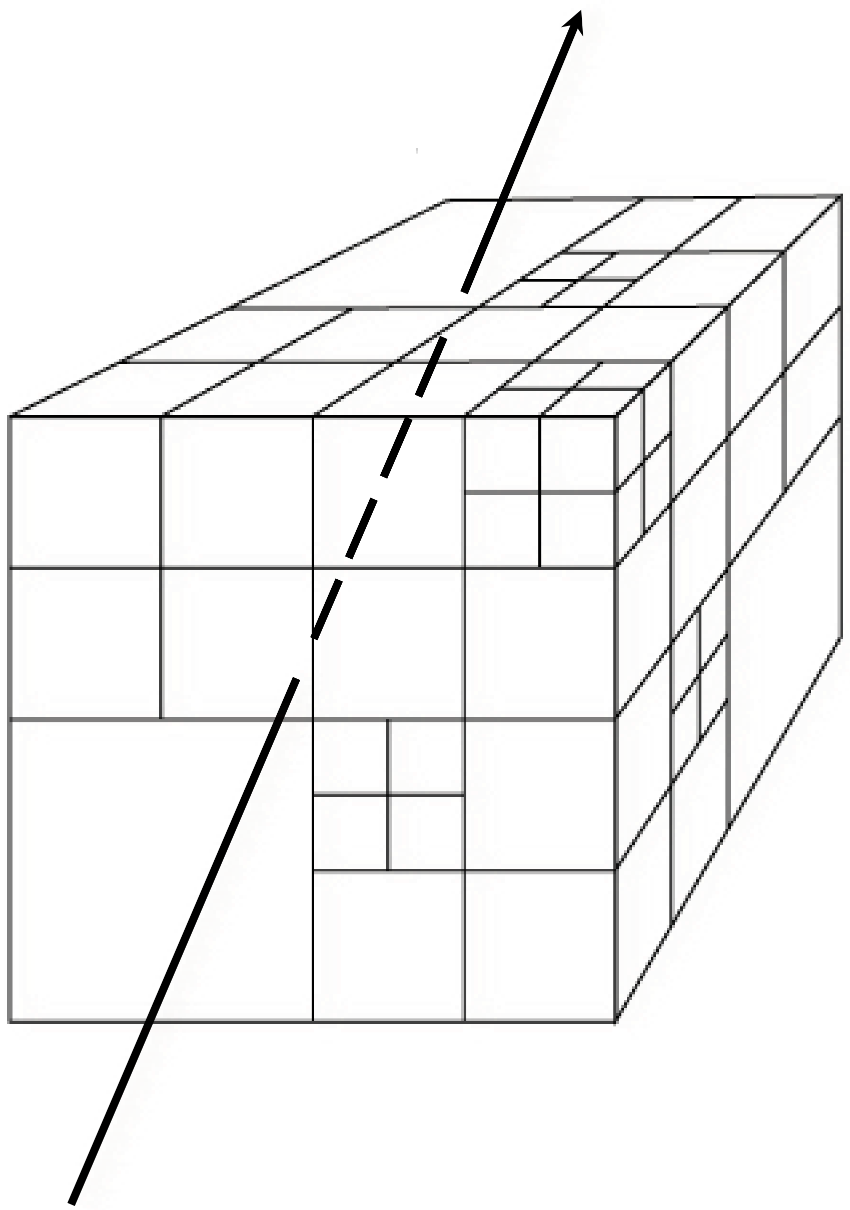
- Preprocessing
  - compute bounding box
  - recursively subdivide cells into 8 equal sub-cells
  - until termination criteria



Octrees

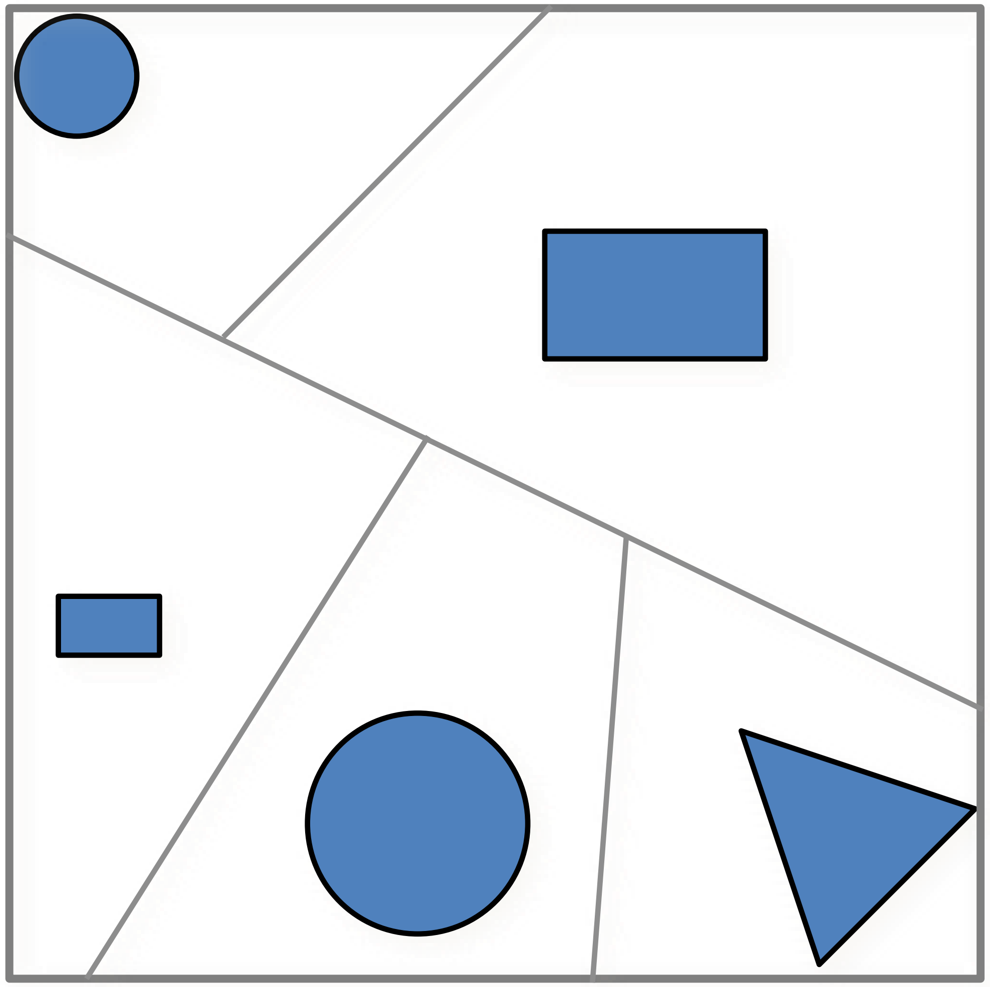
- Traversal
  - Similar to kd-trees
- Easier to implement
- Cheaper costs for
  - Insertion
  - Deletion

- Generally less effective division of space



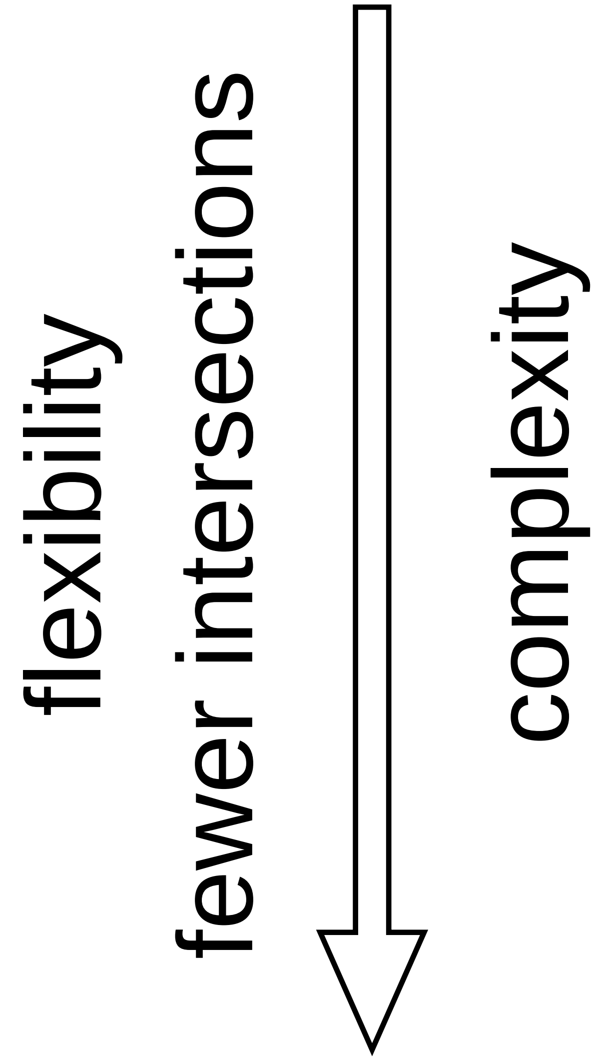
General BSP-Trees

- Preprocessing
  - compute bounding box
  - recursively split space using arbitrary planes



Comparison

- Spatial subdivision based on divide-and-conquer
  - Octree
    - fixed splitting operation
  - Kd-tree
    - fixed plane orientation, variable position & axis
  - BSP tree
    - arbitrary planes



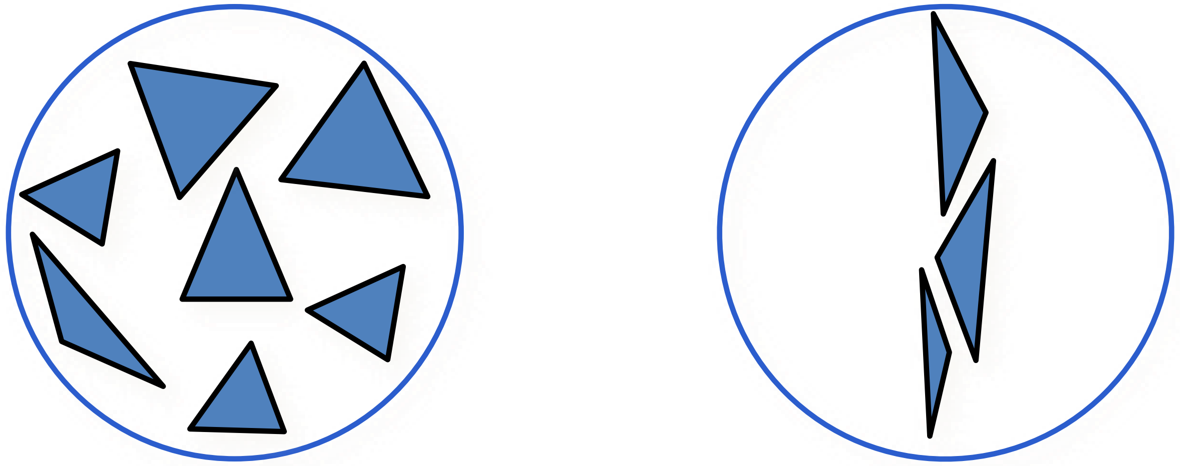
Visual Break

Bounding Volume Hierarchies

- Alternative divide-and-conquer method
- Spatial sorting
  - Decompose **space** into disjoint **regions** & assign objects to regions
- Bounding volumes
  - Decompose **objects** into (overlapping) **sets** & bound using simple volumes for fast rejection

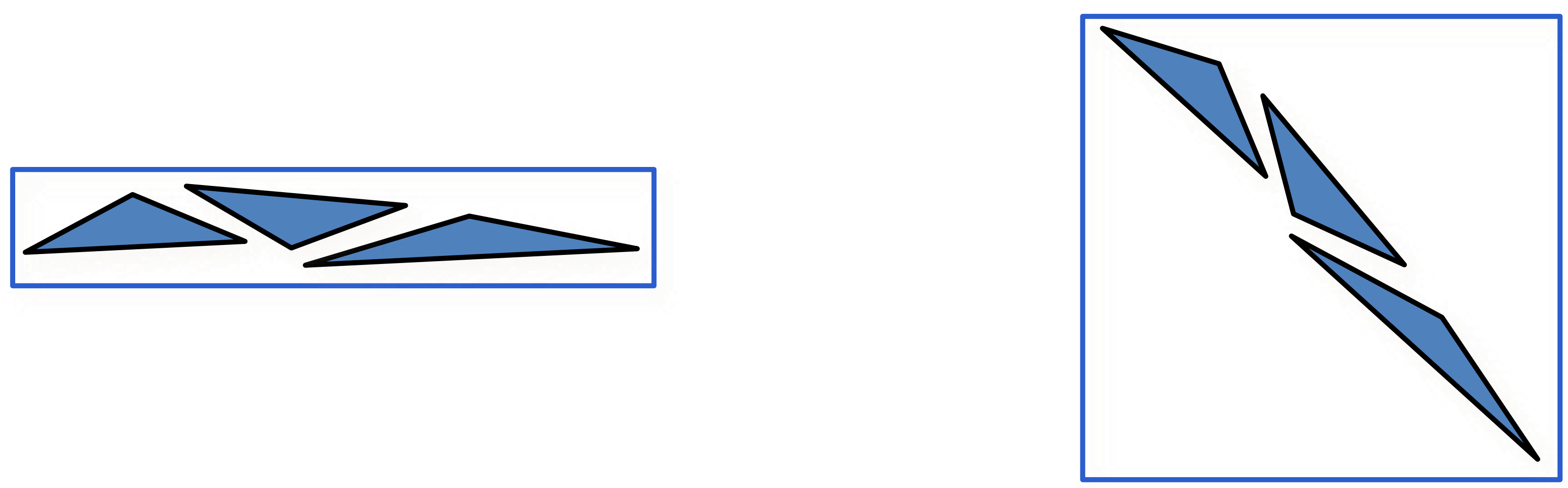
Bounding Volume Hierarchies

- Bounding Volumes
  - Spheres



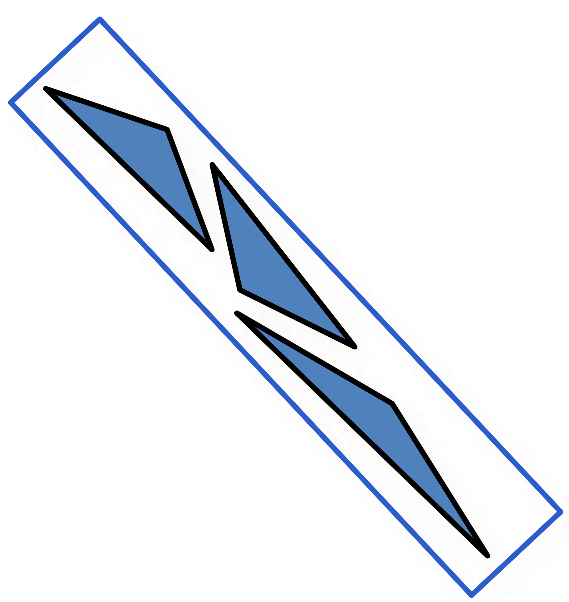
Bounding Volume Hierarchies

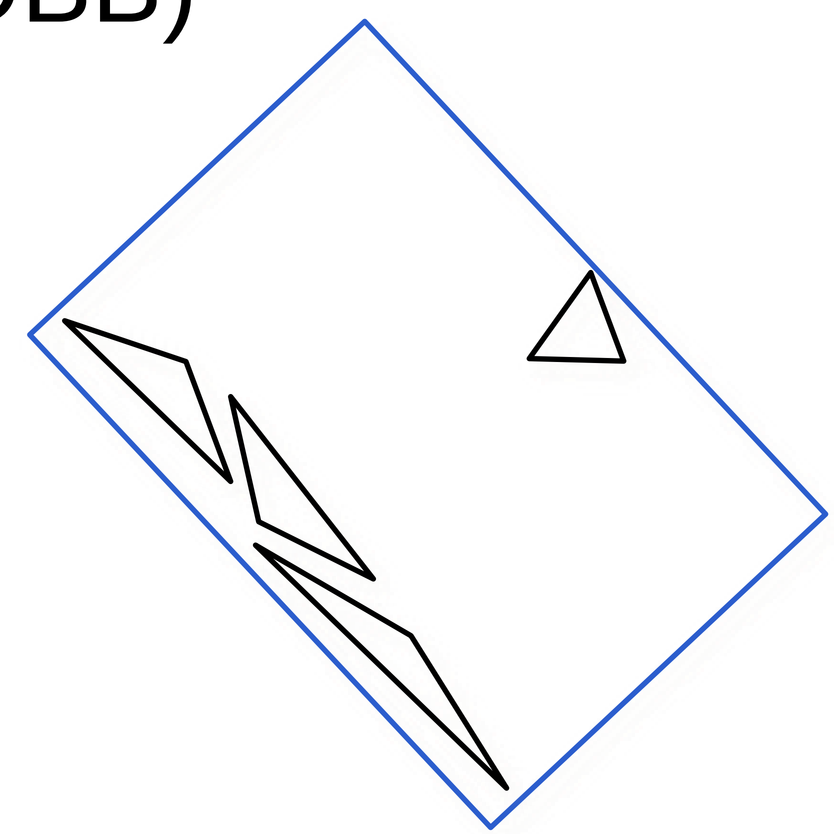
- Bounding Volumes
  - Spheres
  - Axis-aligned bounding box (AABB), most common



Bounding Volume Hierarchies

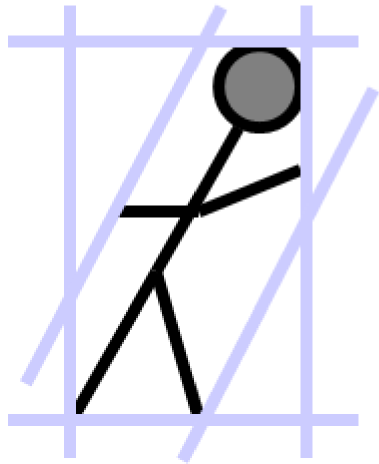
- Bounding Volumes
  - Spheres
  - Axis-aligned bounding box (AABB), most common
  - Oriented bounding box (OBB)



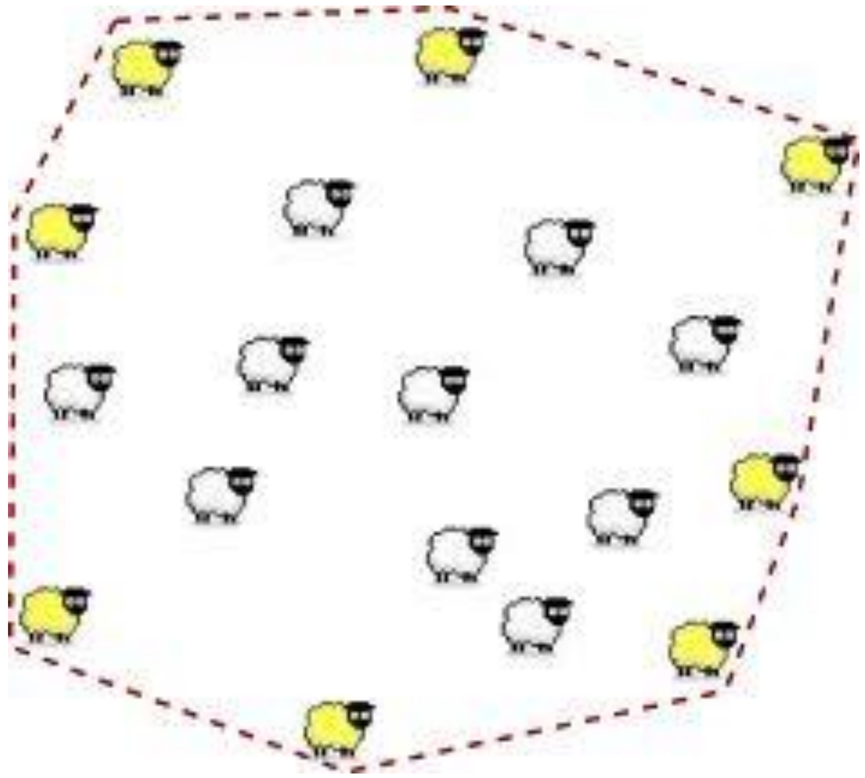


More Bounding Volume Hierarchies

- Bounding Volumes
  - K-discrete orientation polytopes (k-DOPs)
  - Convex hulls, etc.
- Tradeoff:
  - complex shape → tight fit → fewer intersections
  - simple shape → fast intersection



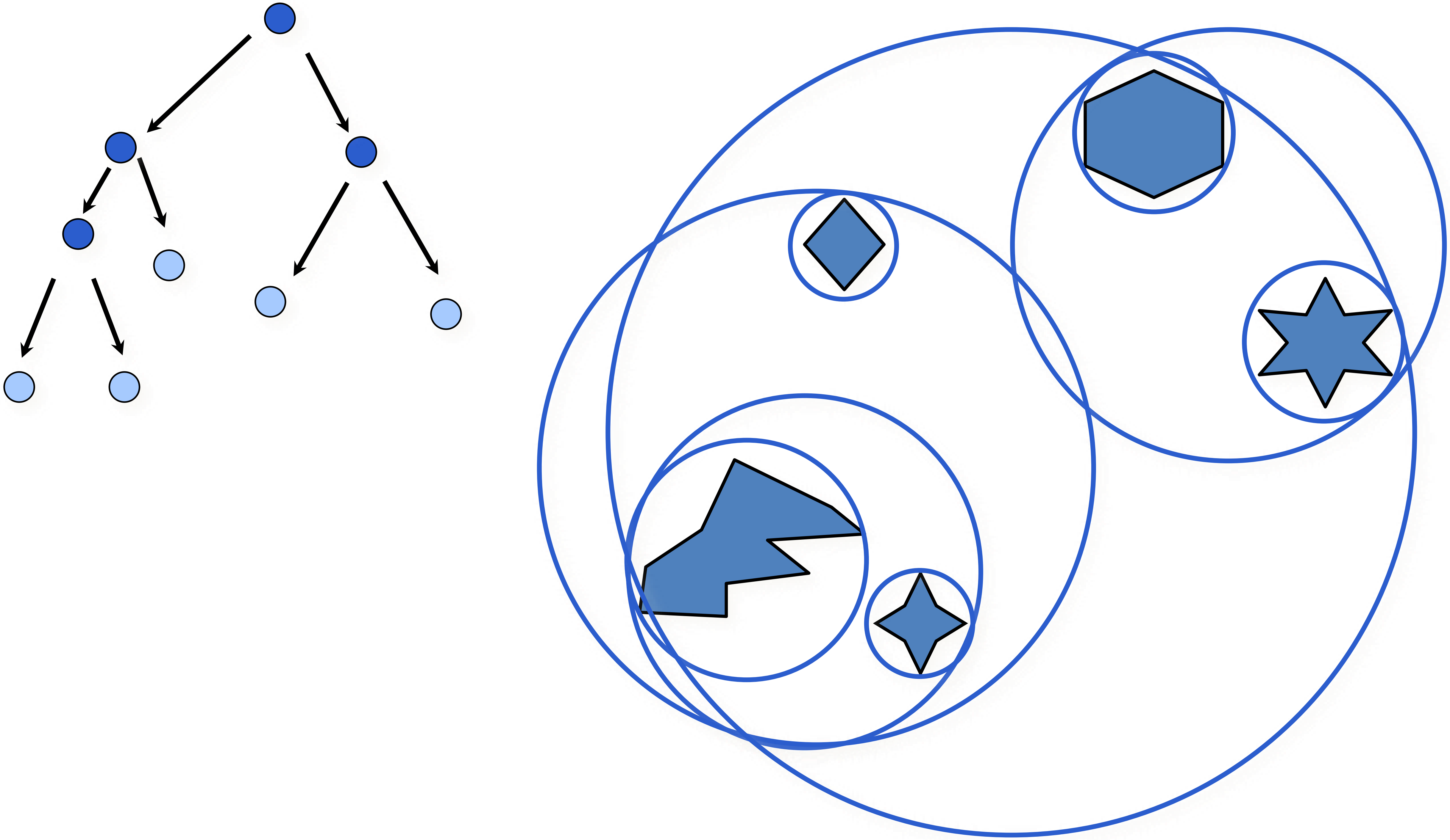
k-DOP



Convex Hull

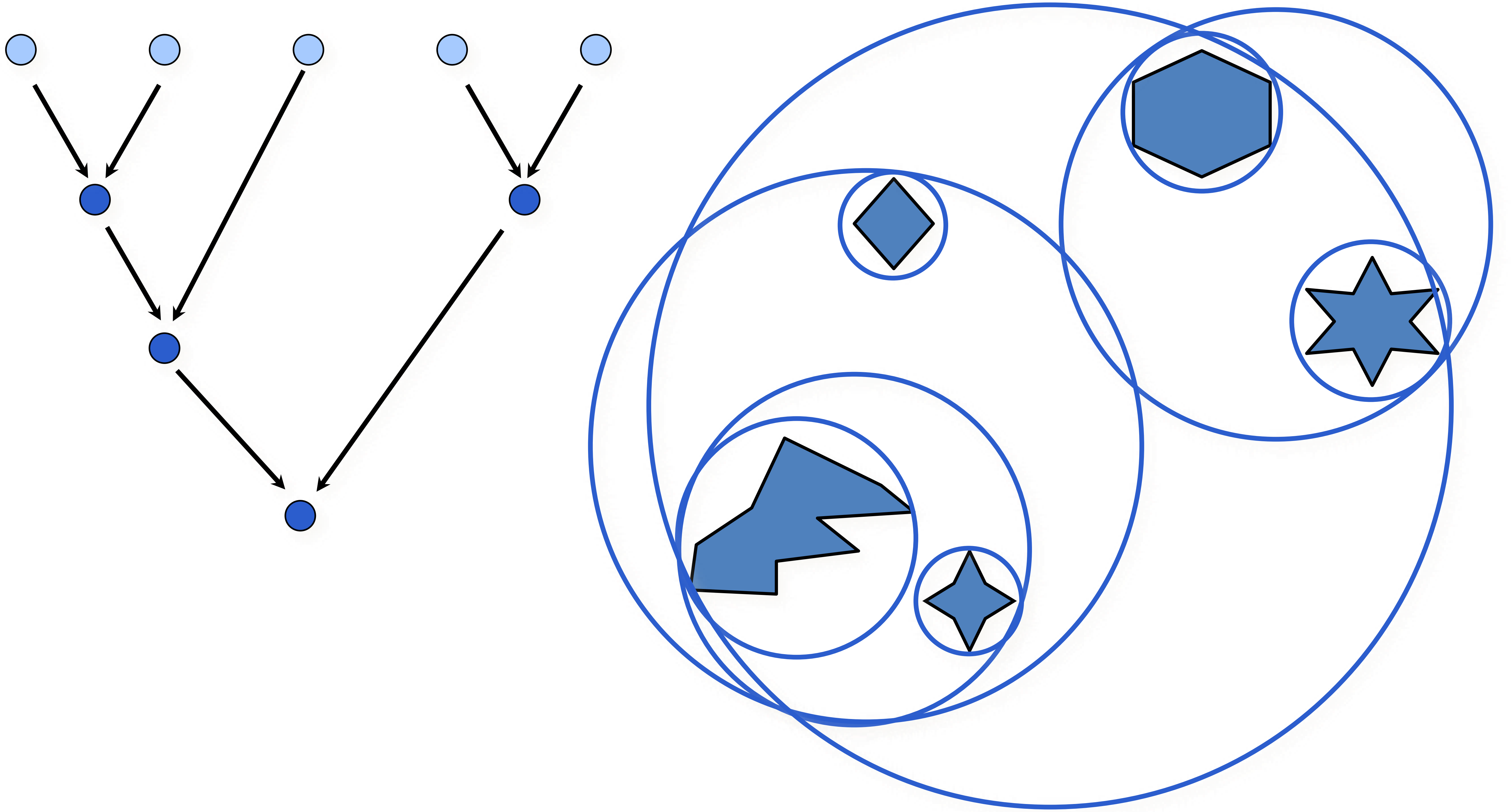
Bounding Volume Hierarchies

- Construction: top down



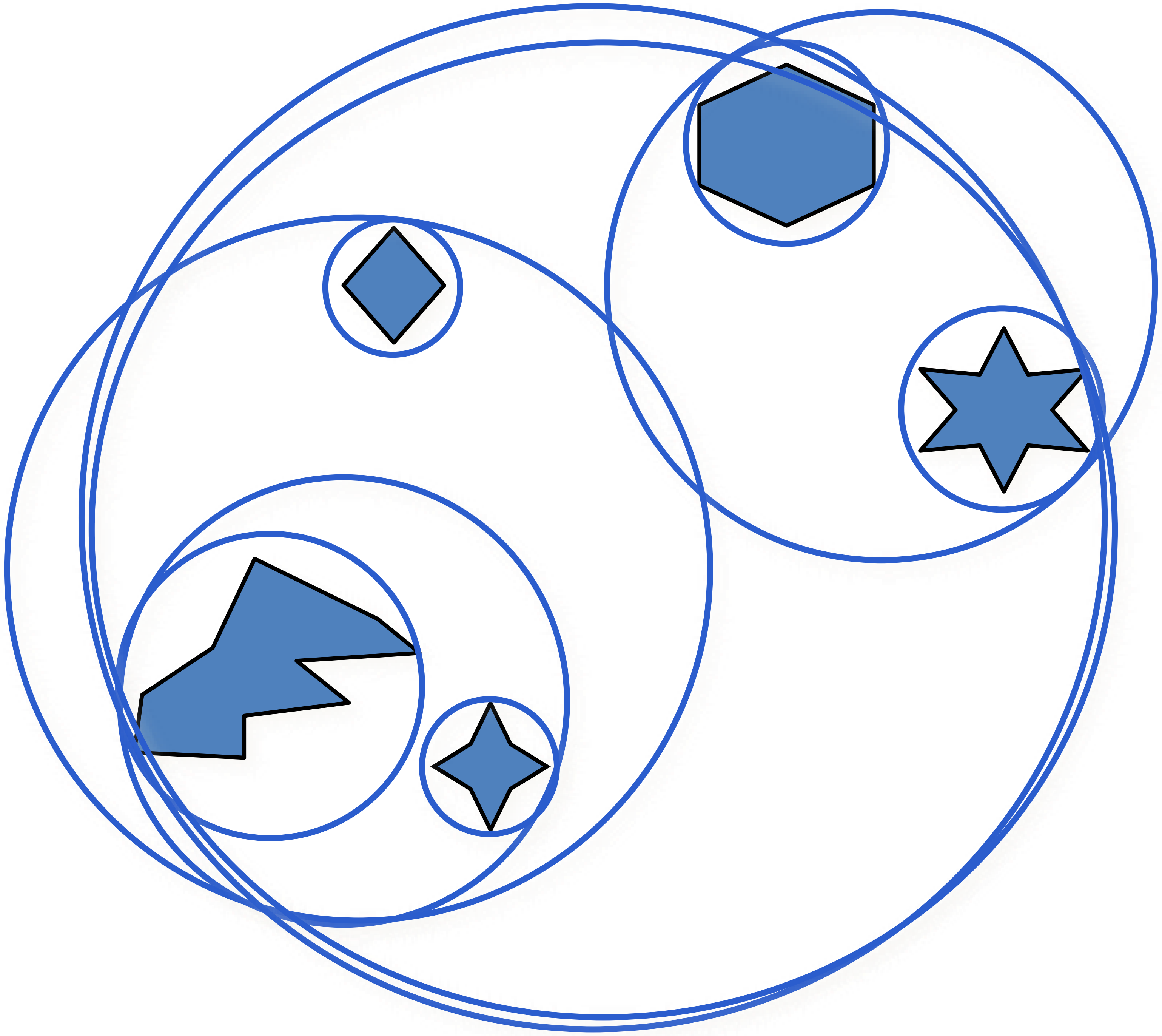
Bounding Volume Hierarchies

- Construction: bottom-up



Bounding Volume Hierarchies

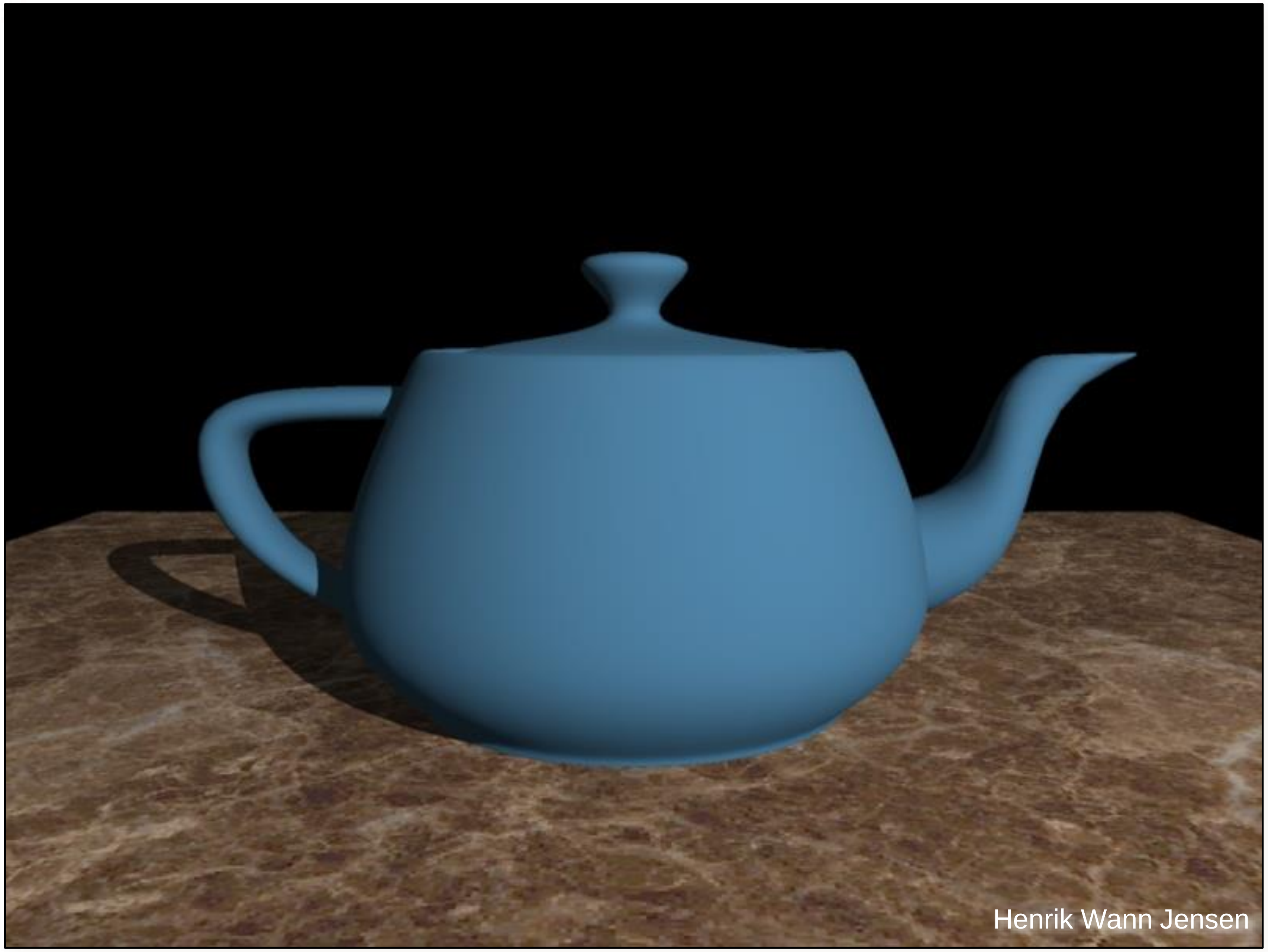
- Construction: insertion



BVH Traversal

```
void intersectBVH( ray, &hit ) {
    if ( boundingBox.hit( ray ) ) {
        if (leaf)
            leaf.intersect( ray, &hit );
        else
            leftChild.intersectBVH( ray, &hit );
            rightChild.intersectBVH( ray, &hit );
    }
}
```

BVH Efficiency



- Brute force: 6321 intersection tests / ray
- Using BVH: 2.6 intersection tests / ray

Summary

- Spatial decomposition
  - inserts objects into **disjoint spatial regions**
  - top-down construction

- Object decomposition
  - partitions objects into **disjoint sets**
  - bounding volumes may **overlap**!
  - bottom-up, or top-down construction

Super Optimizations

- Lots of opportunity for extra optimizations:
- Carefully written inner loop
  (avoid recursion, use your own stack!)
- Compact data structures
  - Ensure small memory footprint for each node
  - Don't store unnecessary cells
- Trace packets of rays
  - 4 or more rays at a time (exploit SSE, etc)
- Thread-level parallelism
- much more

Compact Data Structures

- A KD-tree node in 25 bytes:

```
struct Node
{
    int splitAxis;
    float splitPos;
    Node *leftChild, *rightChild;
    bool isLeaf;
    Object *objArray;
    int numObjects;
}
```

What can we do to reduce the size?

Compact Data Structures

- A KD-tree node in 21 bytes:

```
struct Node
{
    int splitAxis;
    float splitPos;
    Node *leftChild;
    bool isLeaf;
    Object *objArray;
    int numObjects;
}
```

Compact Data Structures

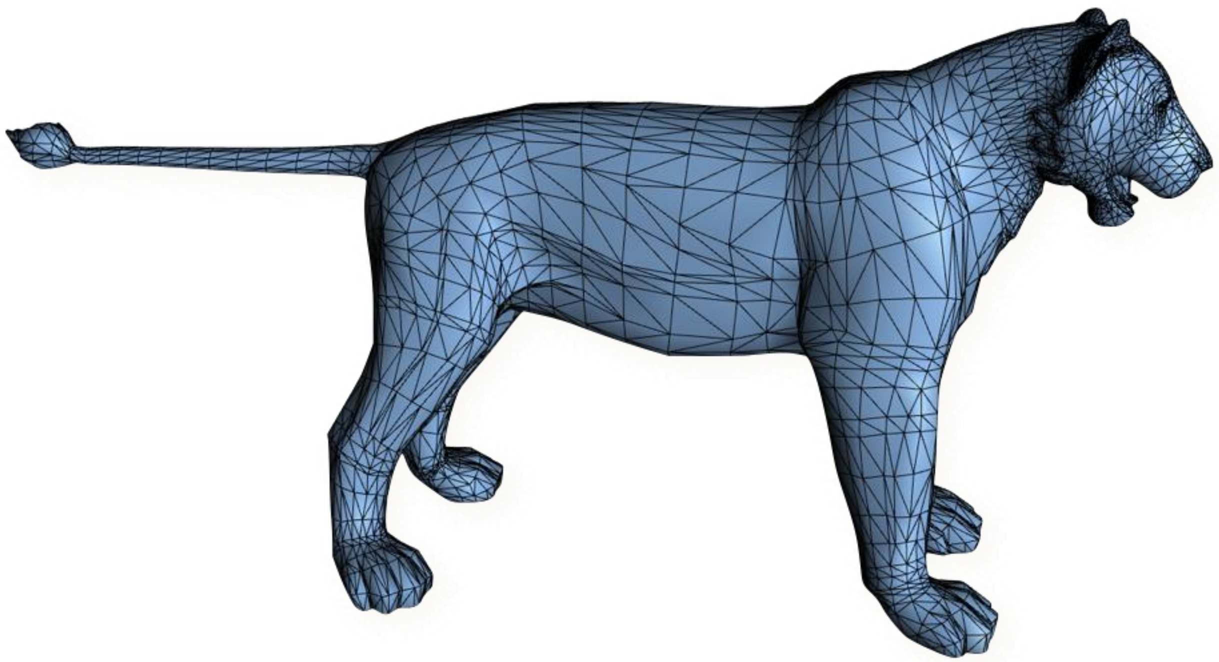
- A KD-tree node in 12 bytes:

```
struct Node
{
    float splitPos;
    void *leftChildOrObjects;
    int flags;  // numObjects, split axis, isLeaf
}
```

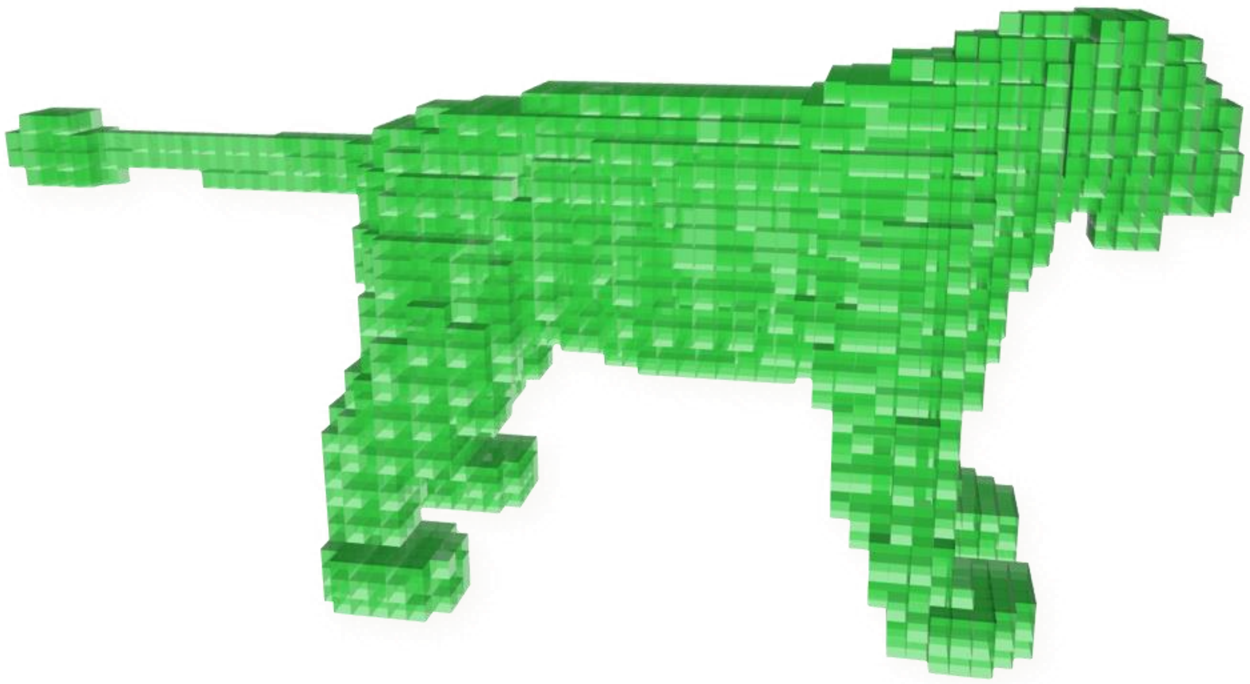
Can be done in 8 bytes!

Compact Data Structures

- Grids
  - many cells may be empty, wasteful to store them
  - store only occupied cells using hashing



input mesh



occupied grid cells

Exploiting Hardware

- caching
- parallelism
- SIMD extensions
- programmable GPUs
- dedicated ray-tracing hardware

Summary

- Key points
  - ray-surface intersections dominate computation effort in ray-tracing
  - spatial pre-sorting significantly reduces ray-surface intersection calculations
    - divide and conquer $O(N) \rightarrow O(\log N)$
  - How to decide which is best?
    - uniform grids, hierarchical grids, kd-trees, bsp-trees, bounding volume hierarchies, ...

Obtaining and using Meshes

- Triangle mesh & texture resources
  - Stanford 3D Scanning Repository
  - NASA 3D Resources
  - Wojciech Jarosz's links page

- Mesh conversion/editing software
  - Blender
  - MeshLab