Spatial Partitioning Data Structures

A Quick Calculation

Number of pixels on screen (1080P):

- 1920 x 1080 = 2,073,600

A Quick Calculation

Number of pixels on screen (1080P):

- 1920 x 1080 = 2,073,600

Number of triangles

- ~millions

Number of ray-triangle intersections:

- ~10^12 intersections per frame

A Quick Calculation

Number of pixels on screen (1080P):

- 1920 x 1080 = 2,073,600

Number of triangles

- ~millions

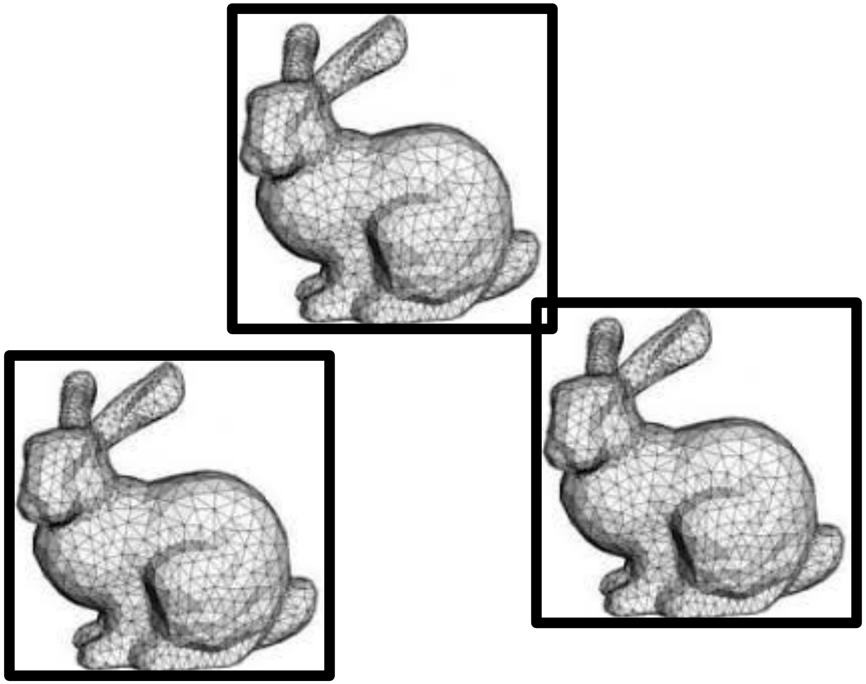
Number of ray-triangle intersections:

- ~10^12 intersections per frame

Now add antialiasing, shadow rays, reflection rays, ……

Bounding Boxes

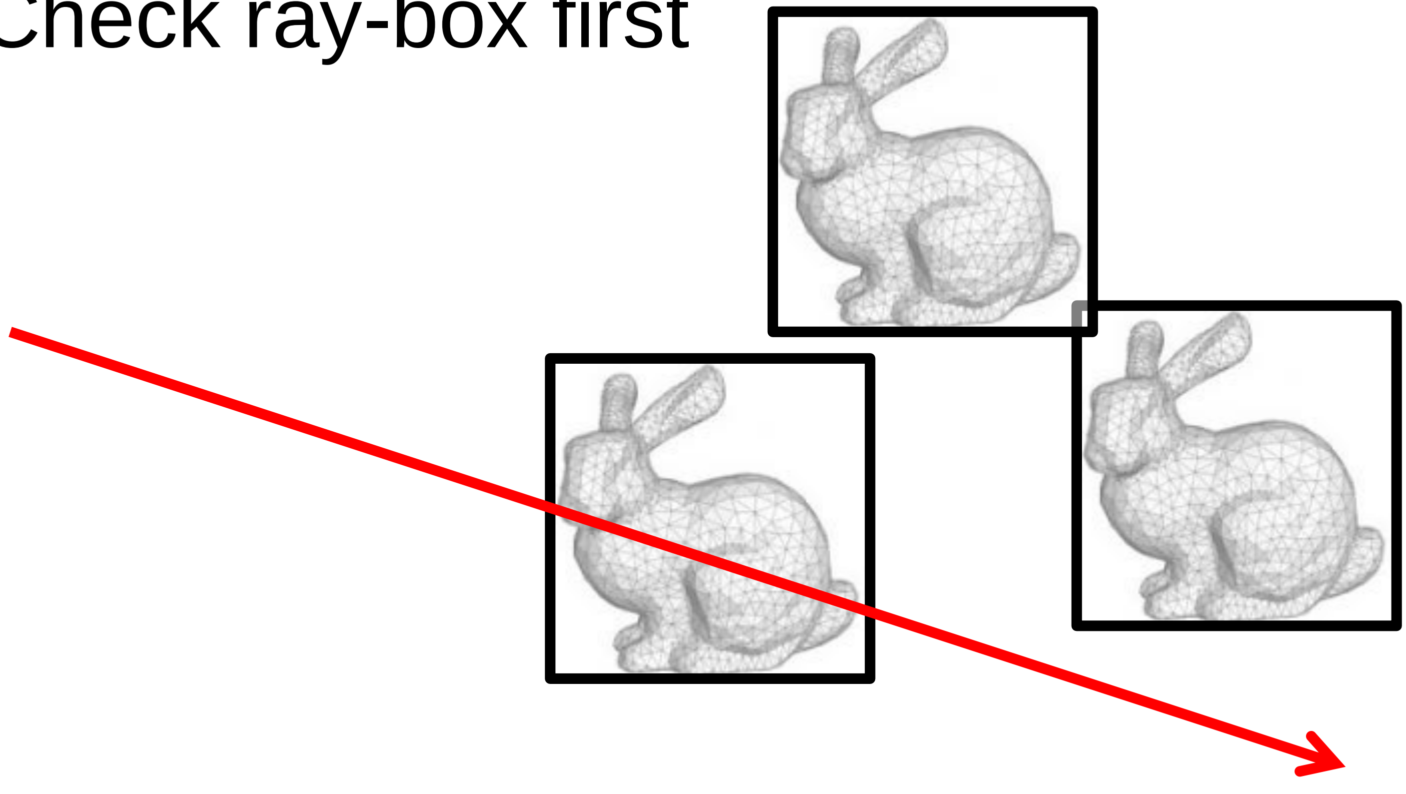
Fit boxes around objects



Bounding Boxes

Fit boxes around objects

Check ray-box first

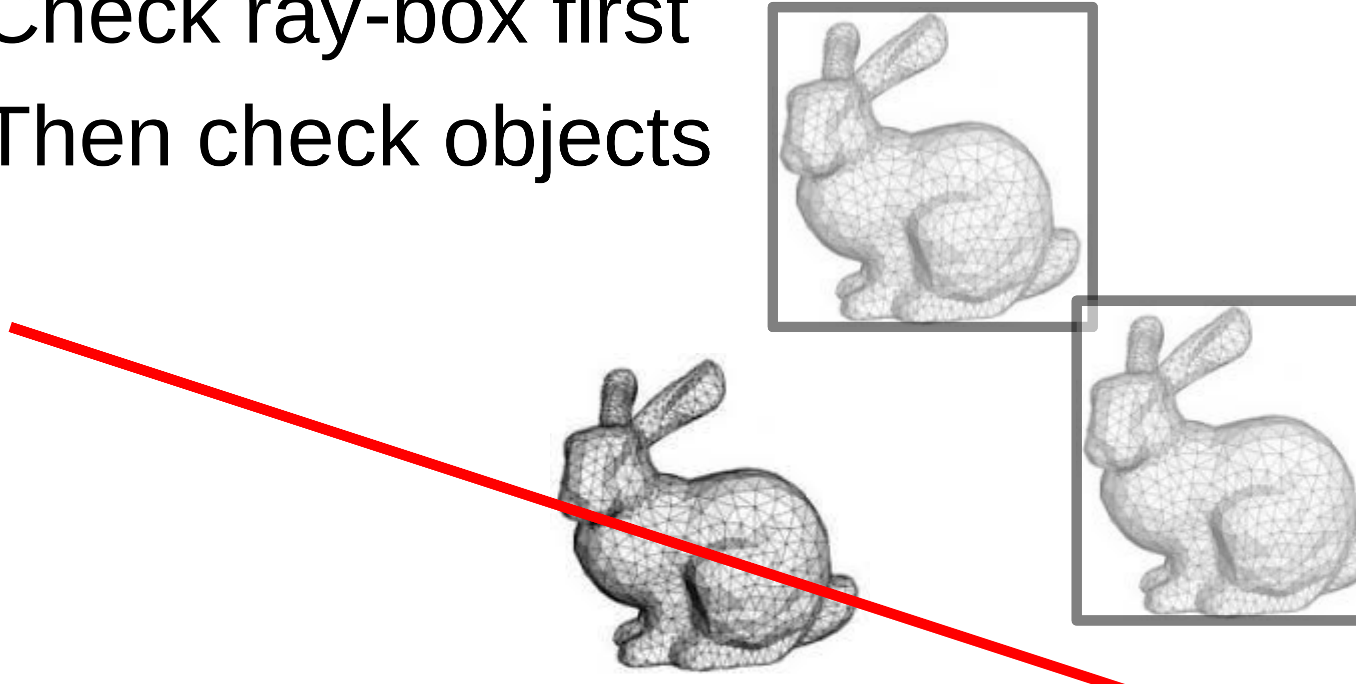


Bounding Boxes

Fit boxes around objects

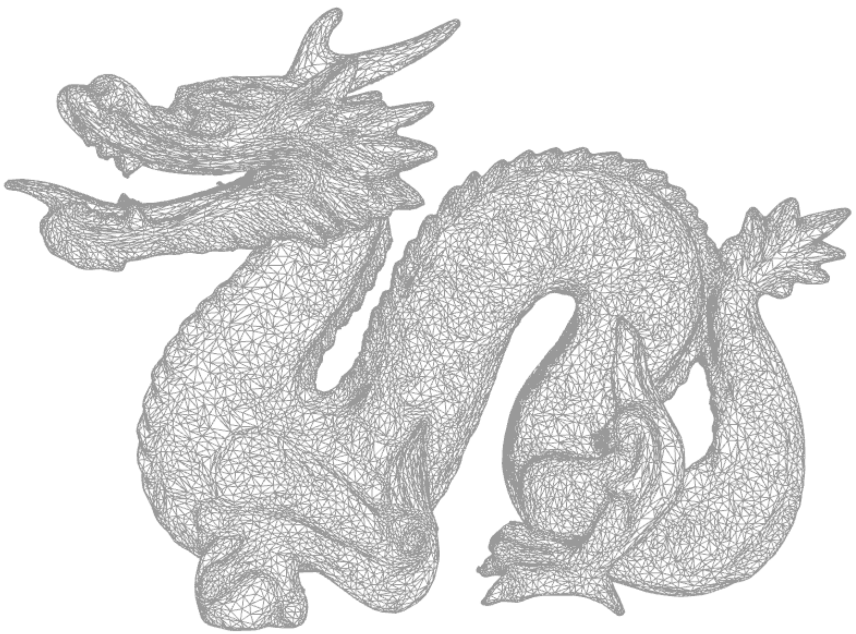
Check ray-box first

Then check objects



Bounding Boxes

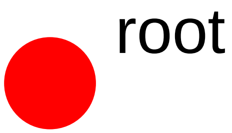
What if we have a single complex object?

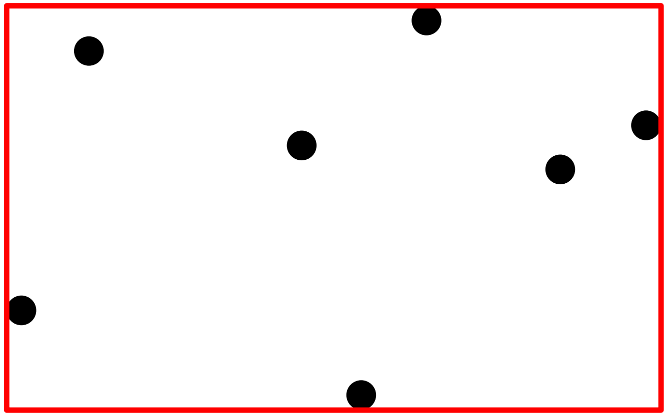


Cut into pieces, treat as separate?

Bounding Volume Hierarchy

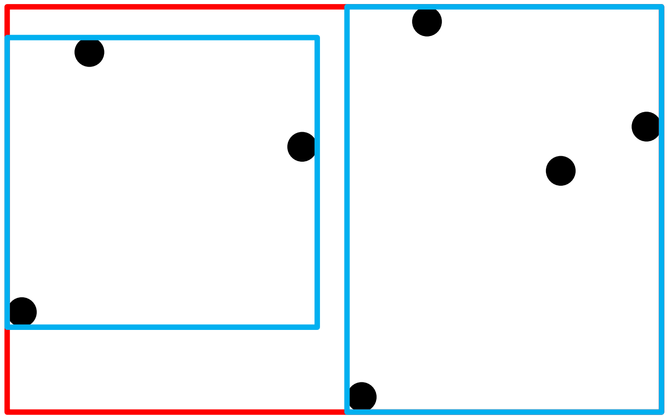
For points:

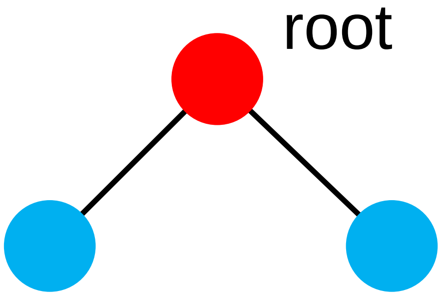




Bounding Volume Hierarchy

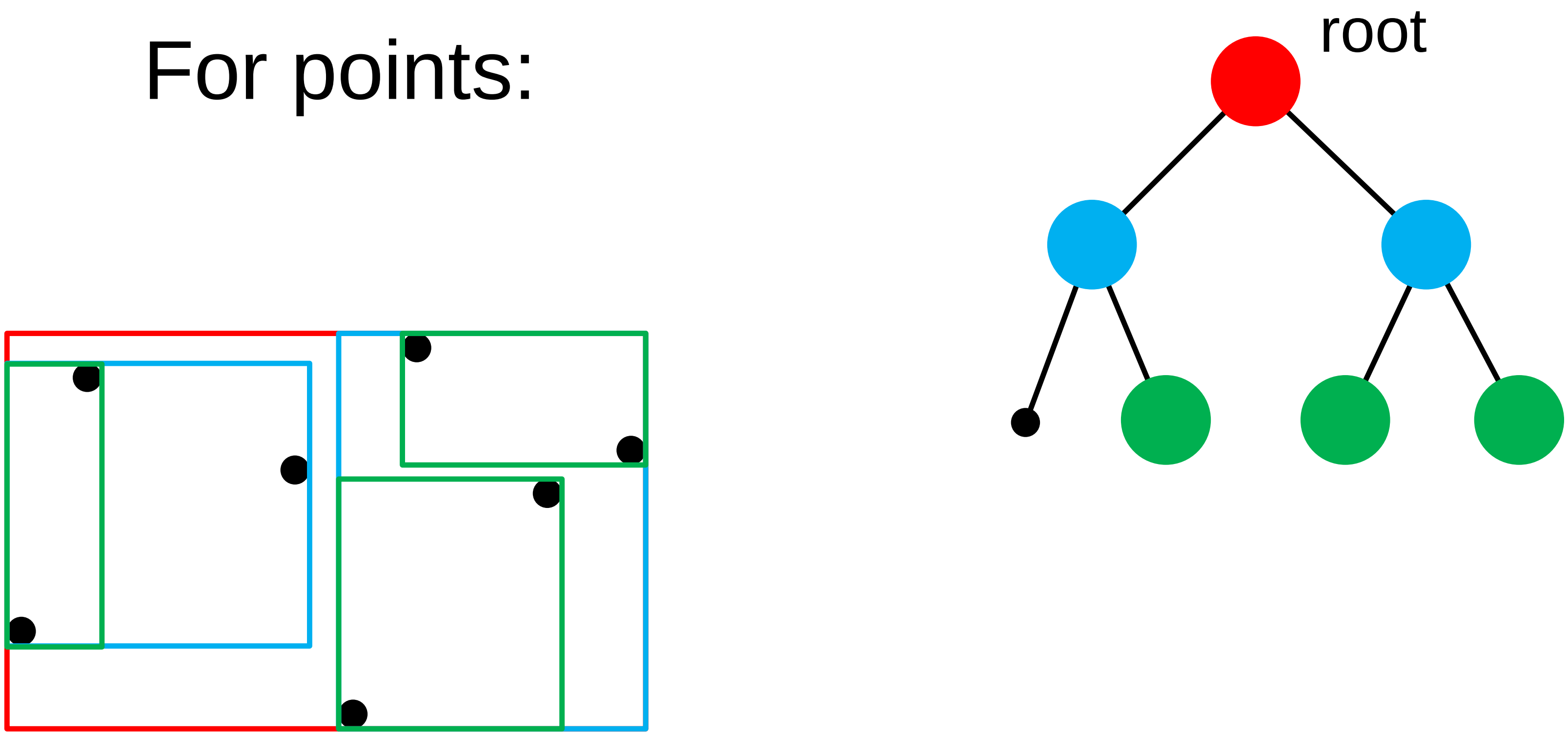
For points:





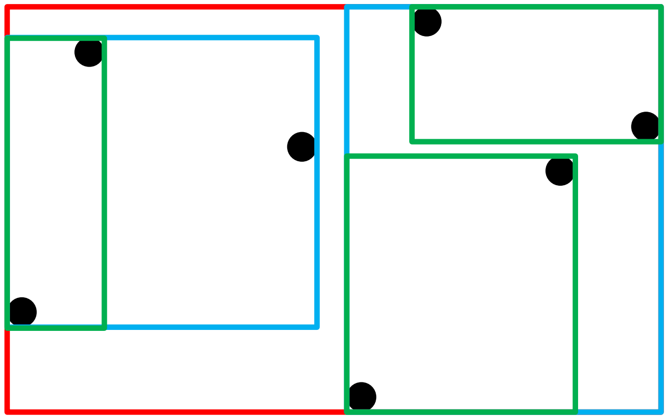
Bounding Volume Hierarchy

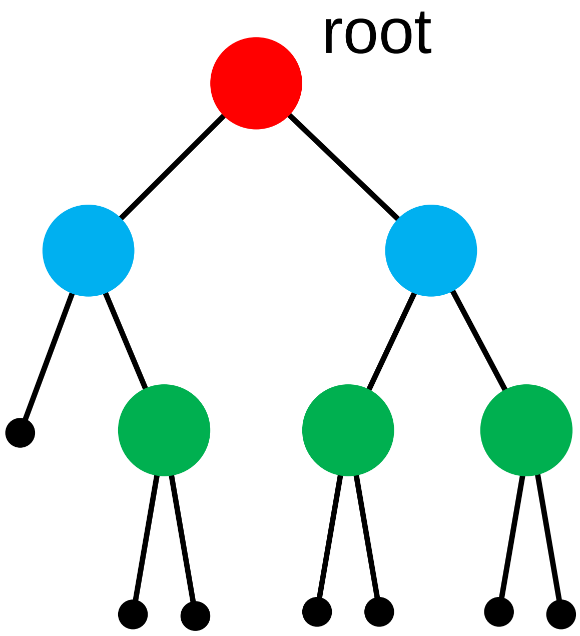
For points:



Bounding Volume Hierarchy

For points:





Bounding Volume Hierarchy

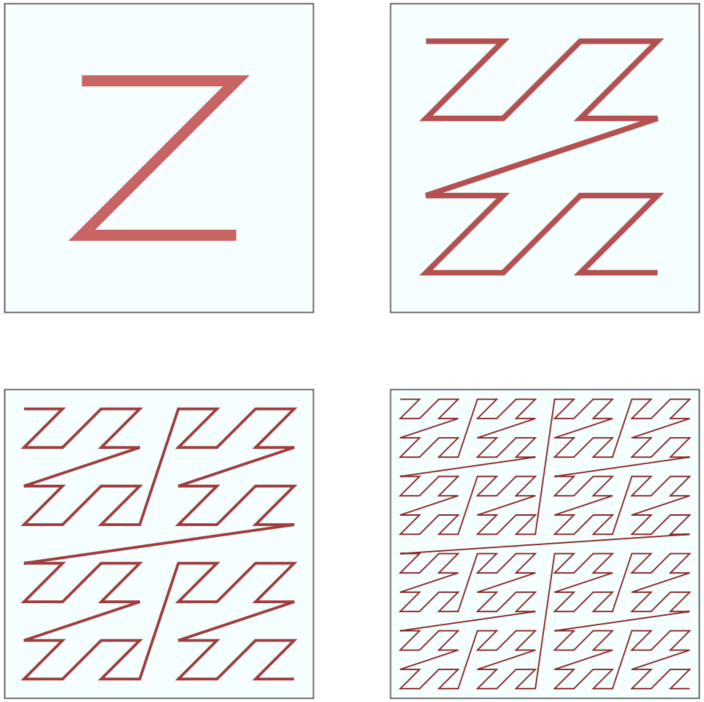
Top-down approach:

```
BuildBVH(points P)
    if P contains one point
        return leaf;
    compute bounding box
    find longest axis
    split points into groups {L, R} along this axis
    return { BuildBVH(L), BuildBVH(R) };
```

Bounding Volume Hierarchy

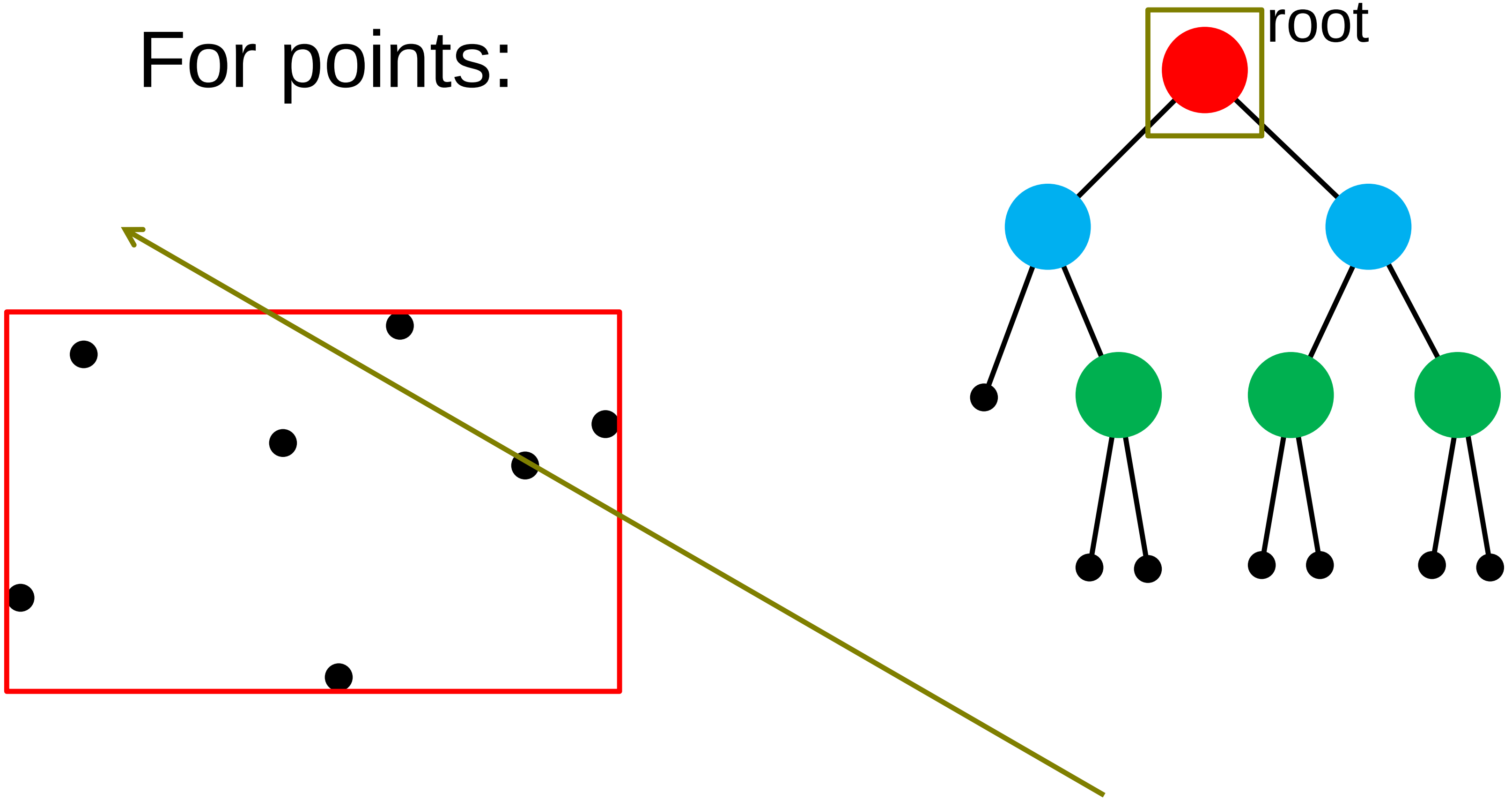
Bottom-up approach (faster, harder):

- sort along space-filling fractal (z-order curve)
- implement using bit fiddling (Morton codes)



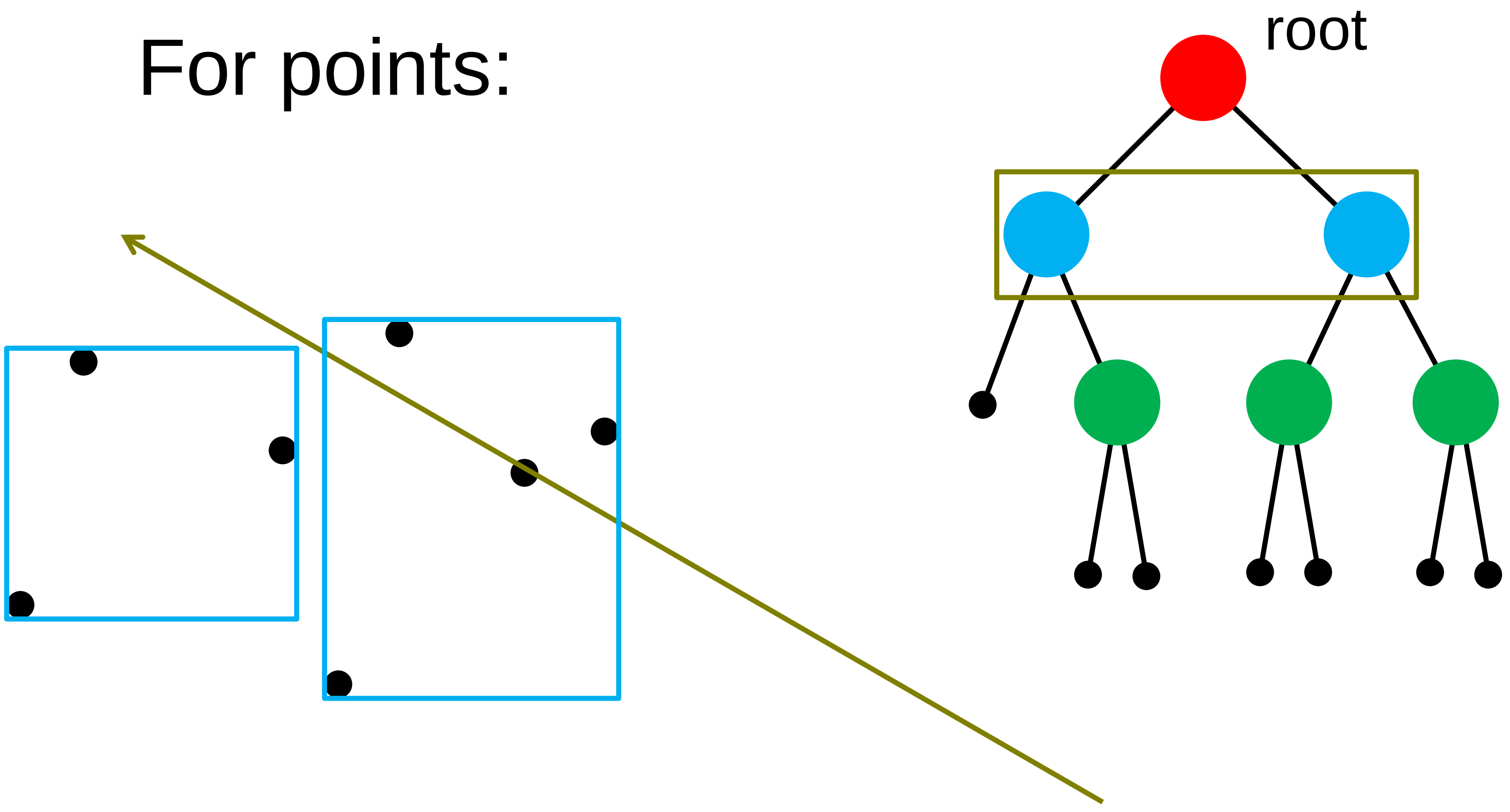
BVH Traversal

For points:



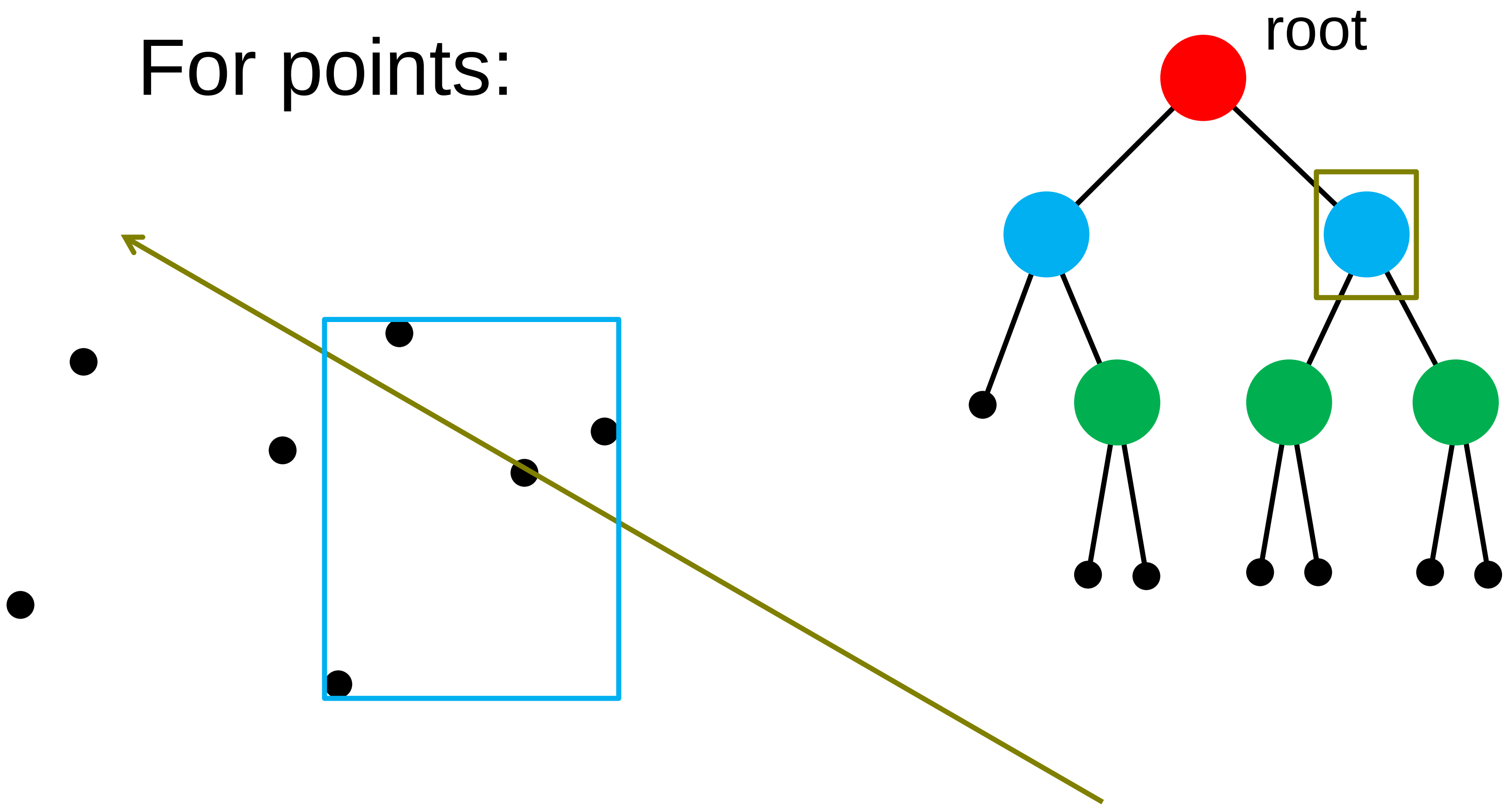
BVH Traversal

For points:



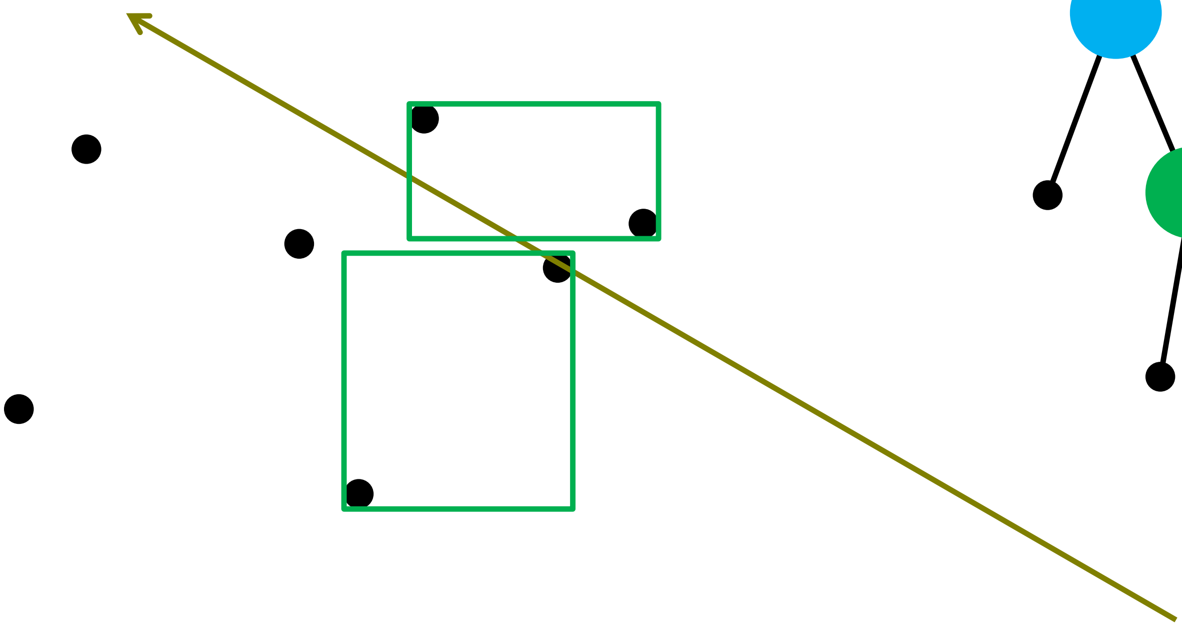
BVH Traversal

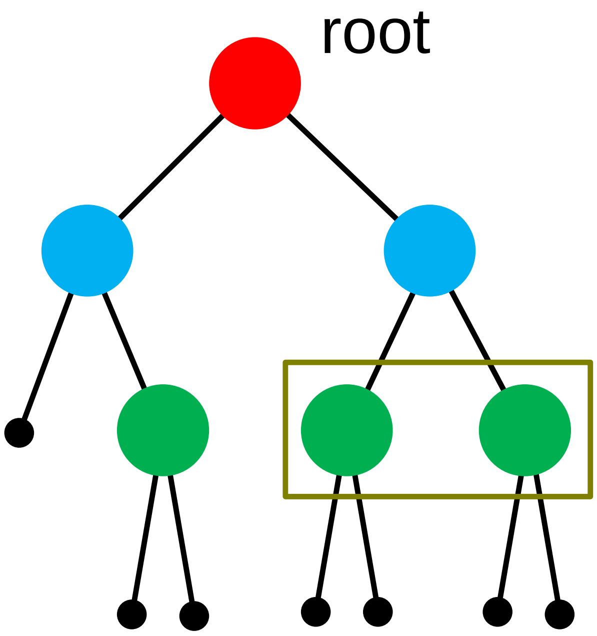
For points:



BVH Traversal

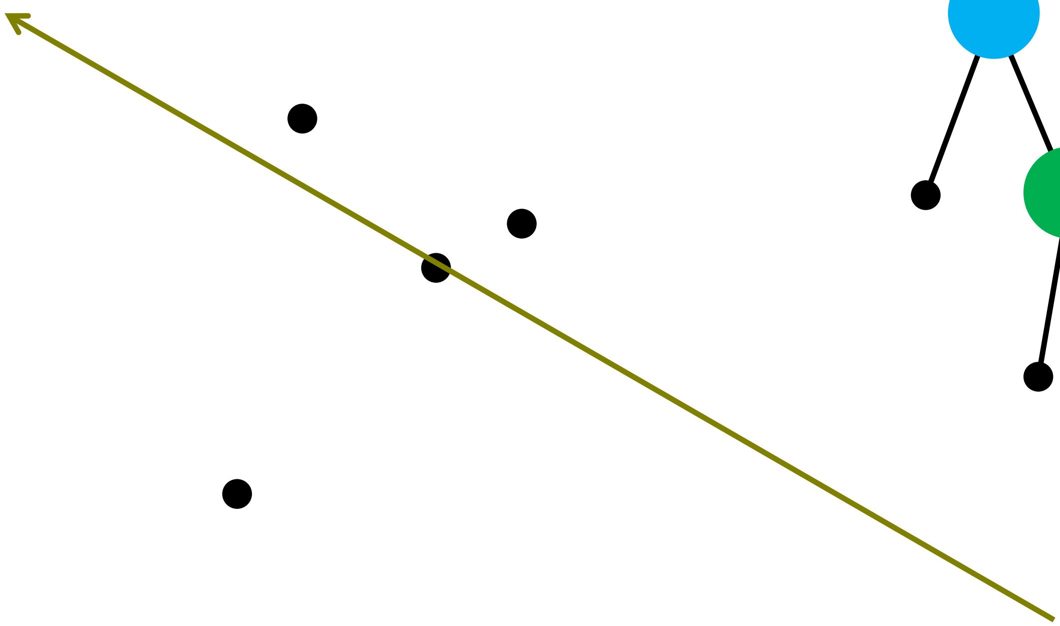
For points:

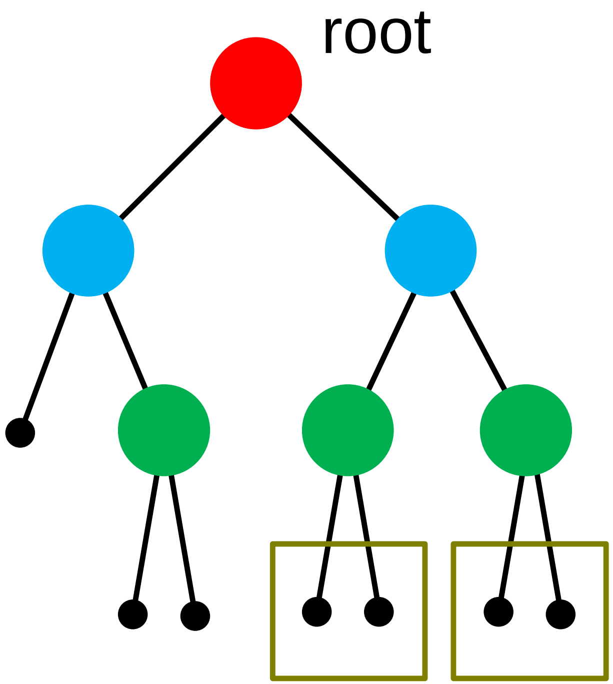




BVH Traversal

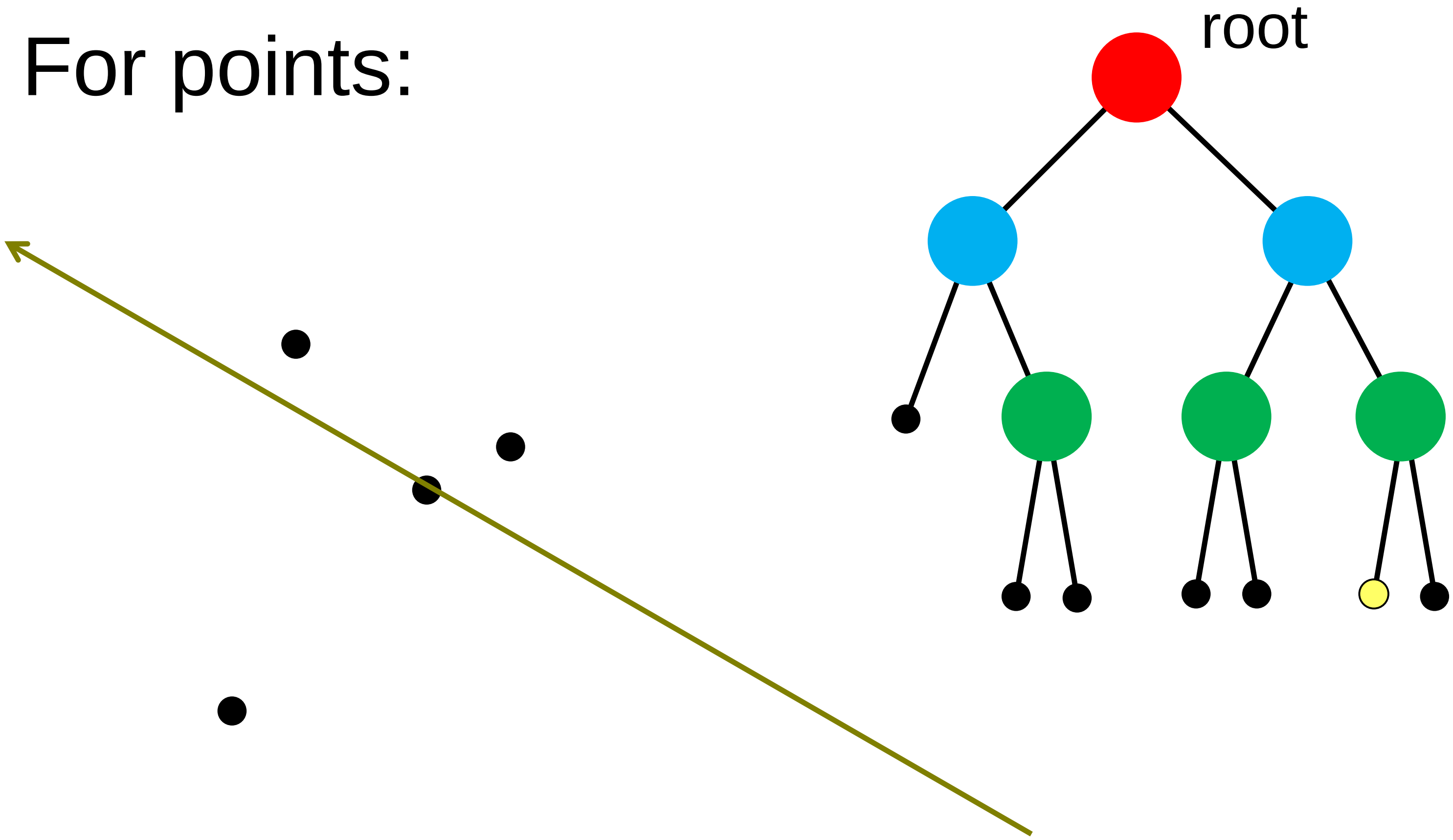
For points:





BVH Traversal

For points:



BVH Analysis

Build time: O(N log^2 N) (top-down)

Traverse time:

BVH Analysis

Build time: $O(N \log^2 N)$ (top-down)

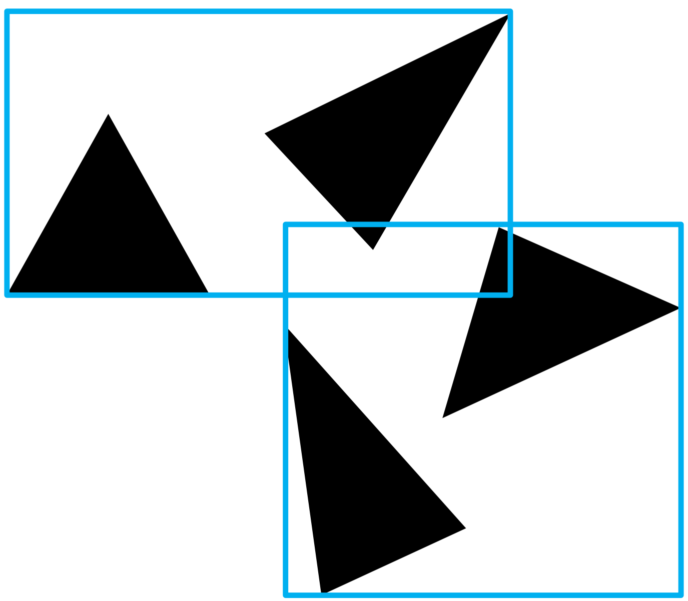
Traverse time:

- worst case: $O(N)$
- typical case: $O(\log N)$

Advanced traversal strategies possible

BVH in Practice

Build around **triangle primitives**



leaves are individual triangles

when building, sort by e.g. triangle center

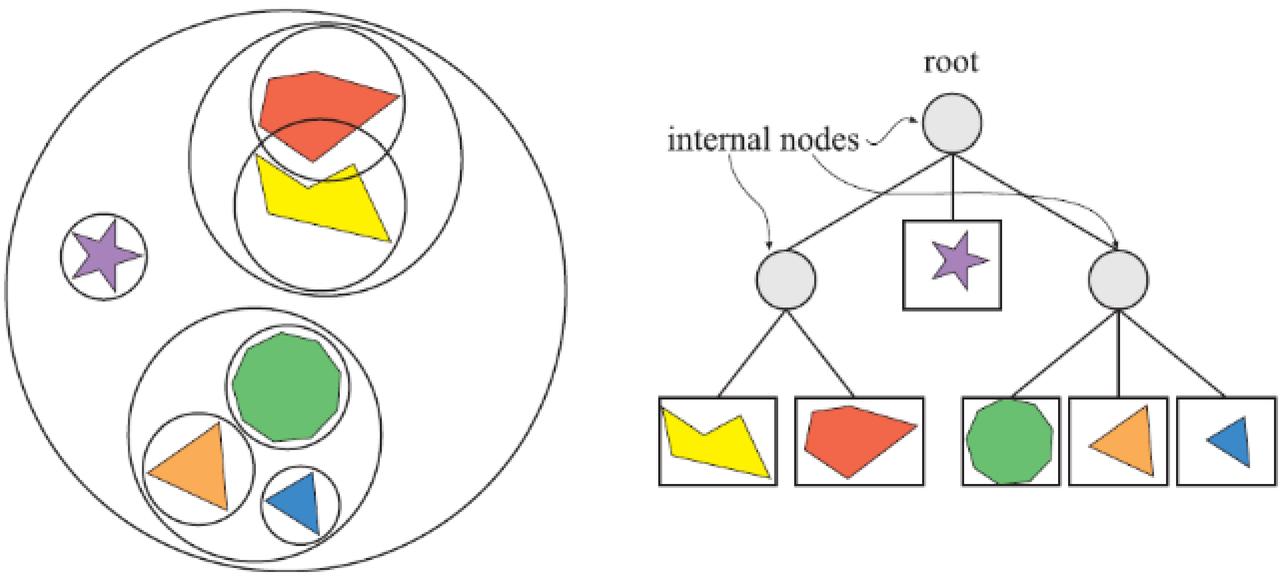
note: nodes can overlap

BVH Node Types

Most typical: **AABB**s

- "axis-aligned bounding boxes"

Other options possible:



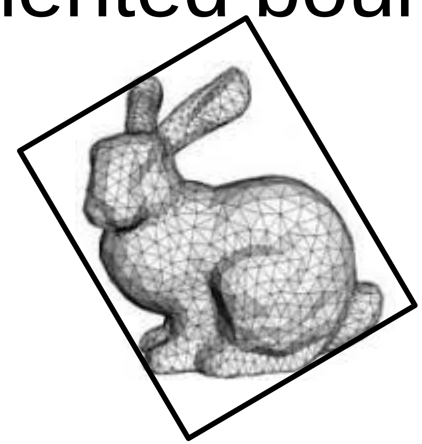
BVH Node Types

Most typical: **AABB**s

- “axis-aligned bounding boxes”

Other options possible:

- sphere trees
- OBBs (oriented bounding boxes)



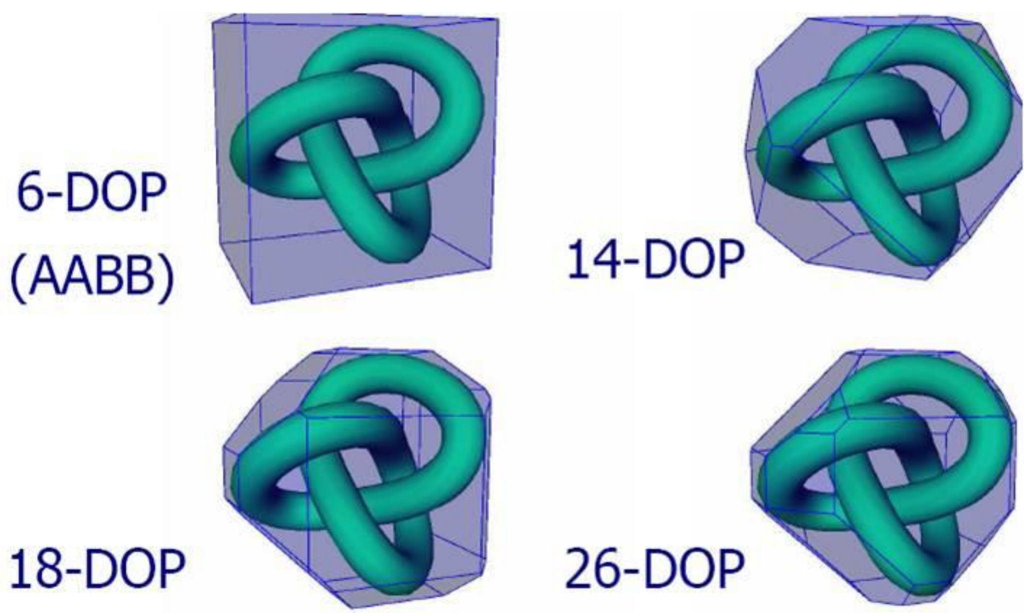
BVH Node Types

Most typical: **AABB**s

- "axis-aligned bounding boxes"

Other options possible:

- sphere trees
- OBBs
- k-DOPs



BVH Node Types

Most typical: **AABB**s

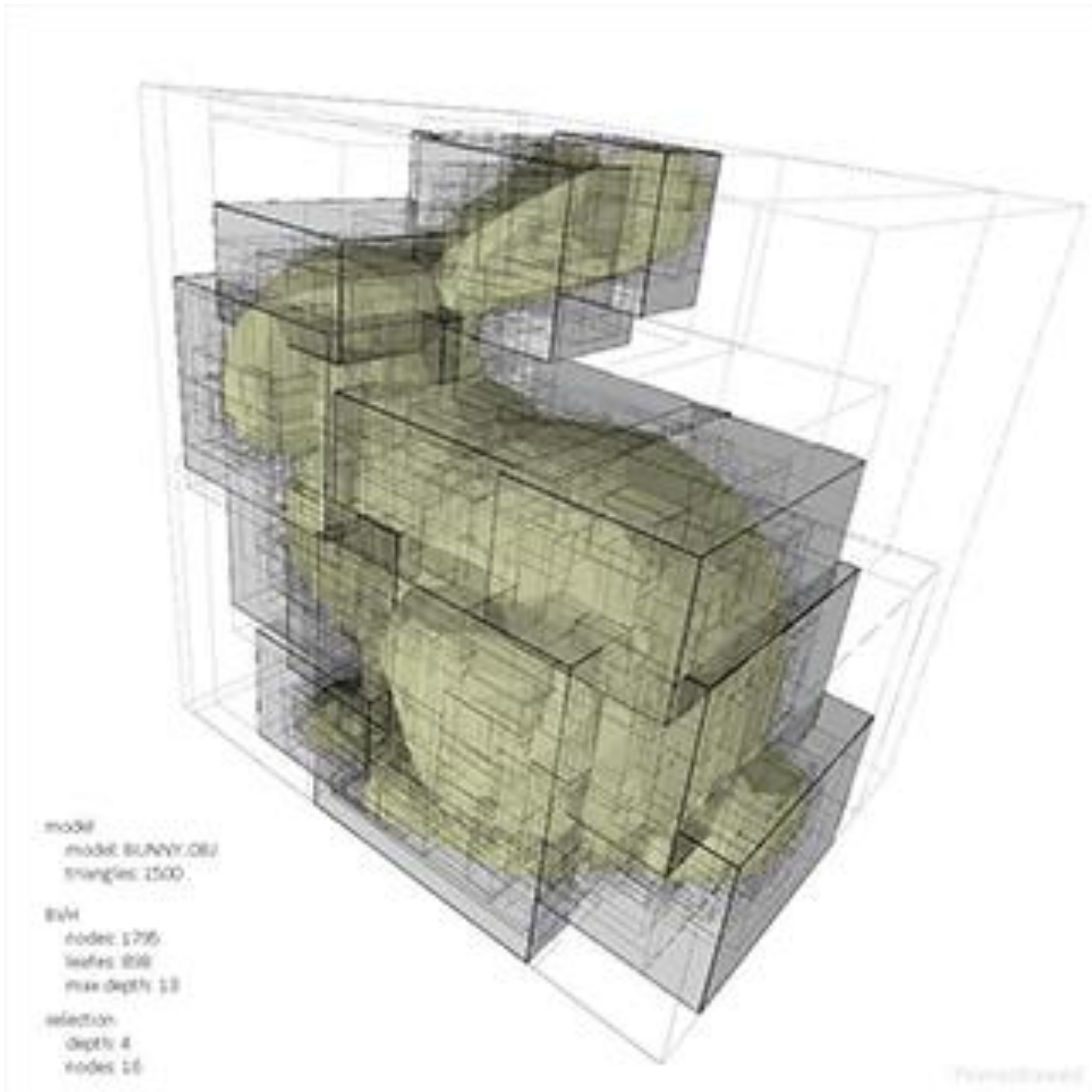
- “axis-aligned bounding boxes”

Other options possible

Complex tradeoff between

- tightness of fit
- traverse cost
- build cost
- memory usage

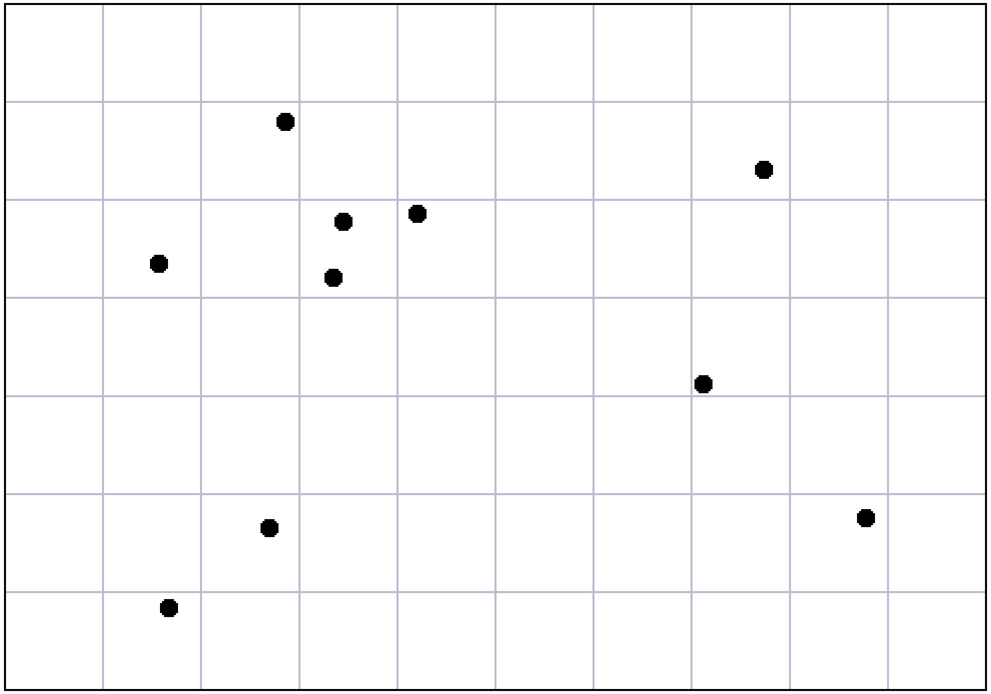
BVH Visualized



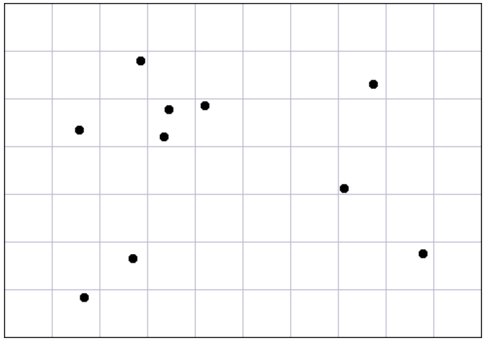
Spatial Hashing

Divide space into **coarse grid**

Each grid cell stores its contents



Spatial Hashing

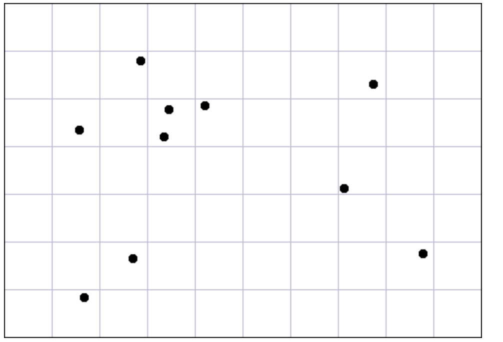


Divide space into **coarse grid**

Each grid cell stores its contents

How to build?

Spatial Hashing



Divide space into **coarse grid**

Each grid cell stores its contents

How to build?

- **hash function** maps points to their cell
- usually very fast (bit twiddling)

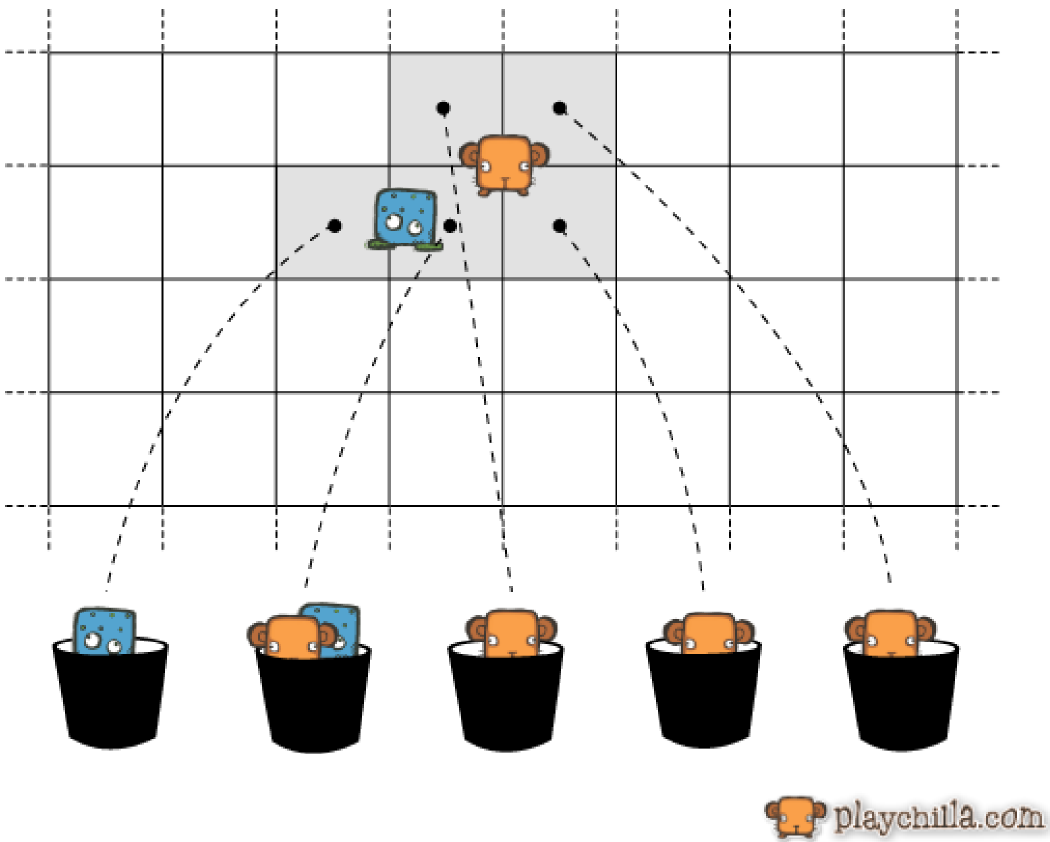
Why useful?

Spatial Hashing

What if primitives aren't point?

Spatial Hashing

What if primitives aren't point?



must **rasterize** objects to grid

object overlaps multiple cells
--> multiple refs

Spatial Hashing

Pros:

- (relatively) simple to build
- simple data structure (array of pointers)

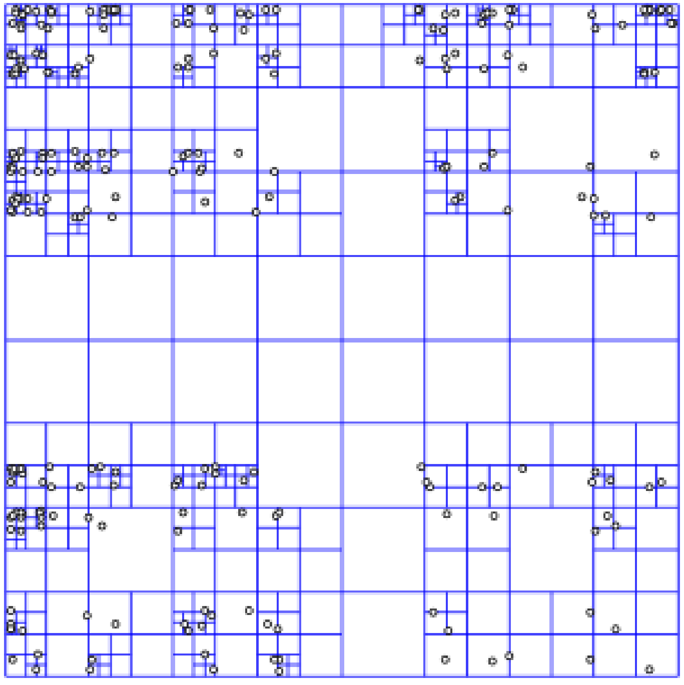
Cons:

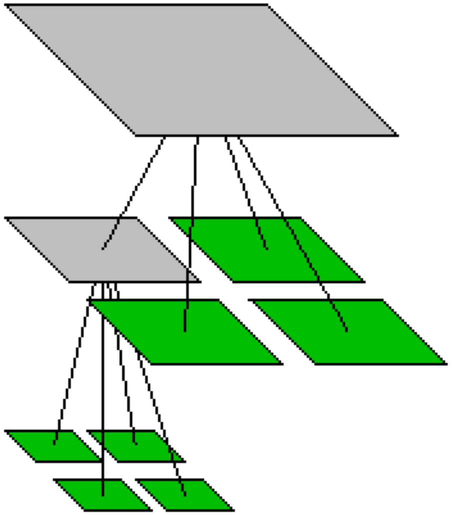
- must pick a good cell size
- works poorly on heterogeneous object distributions

Quadtree

Start with spatial hash

Split crowded cells into child squares

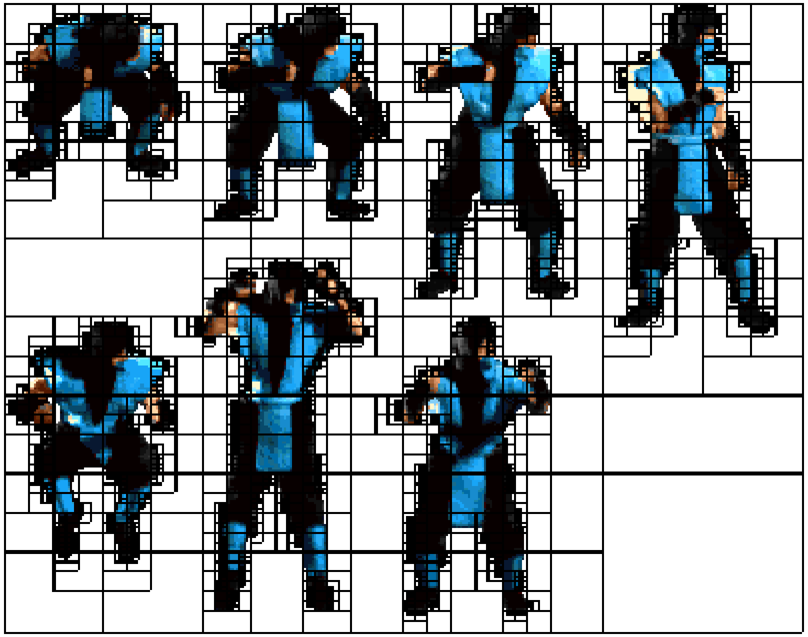


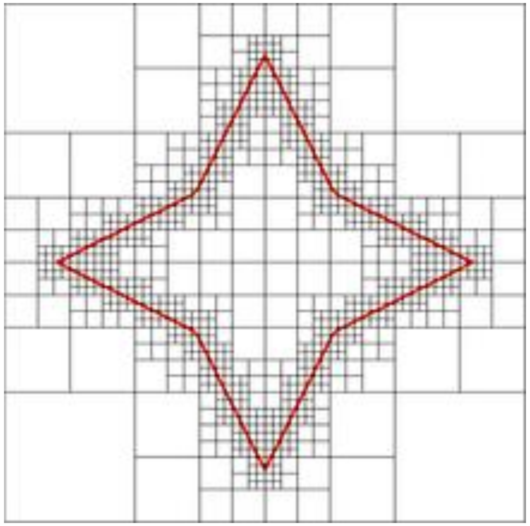


Quadtree

Works also for non-point primitives

Danger – must pick maximum depth





Quadtree

Pros:

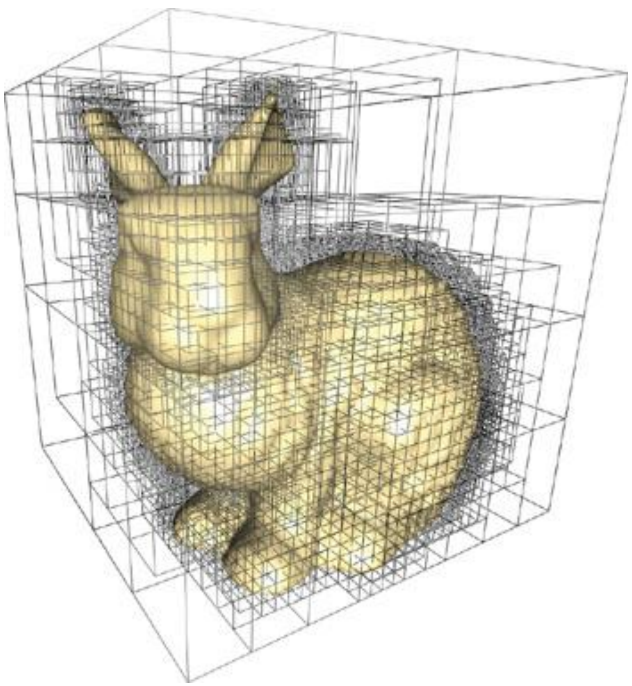
- very space-efficient even for heterogeneous object distributions
- simple to build and traverse (bit tricks often used)

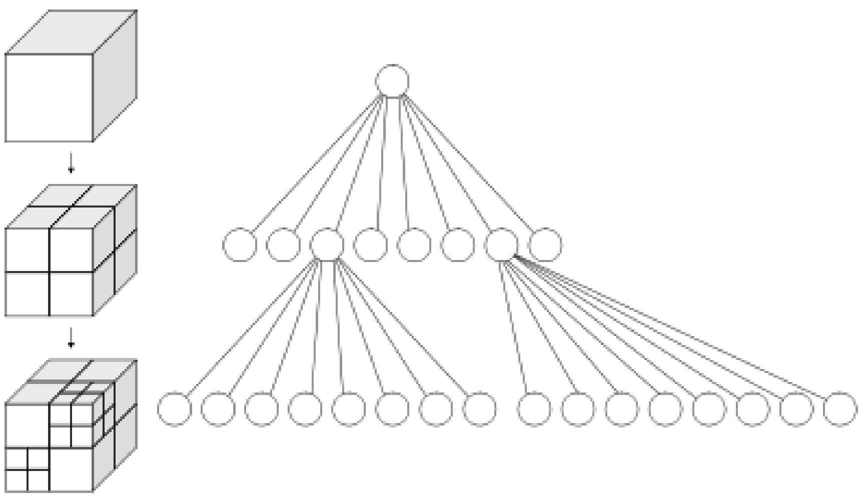
Cons:

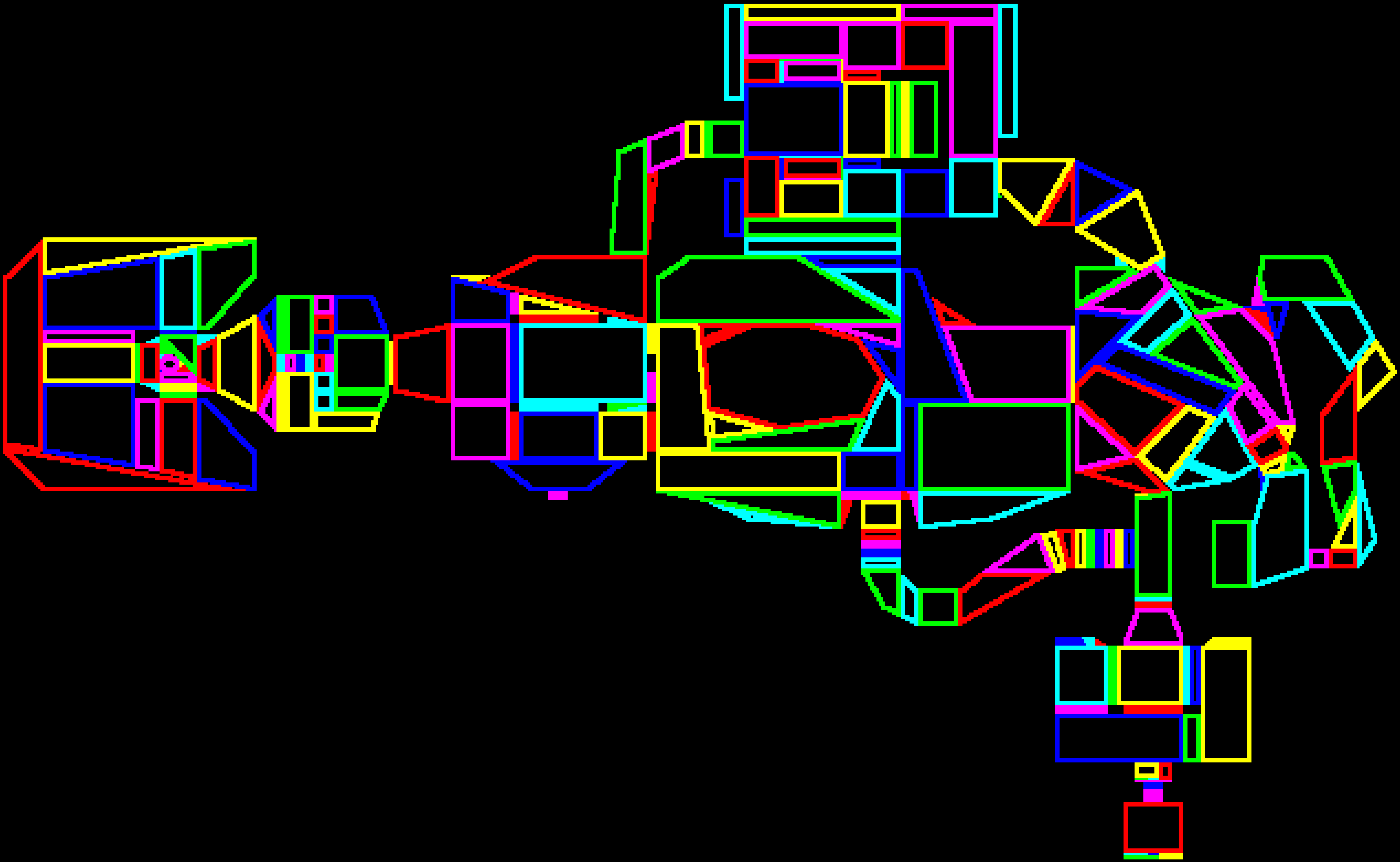
- must pick max tree depth
- tree not balanced

Octree

3D version of quadtree

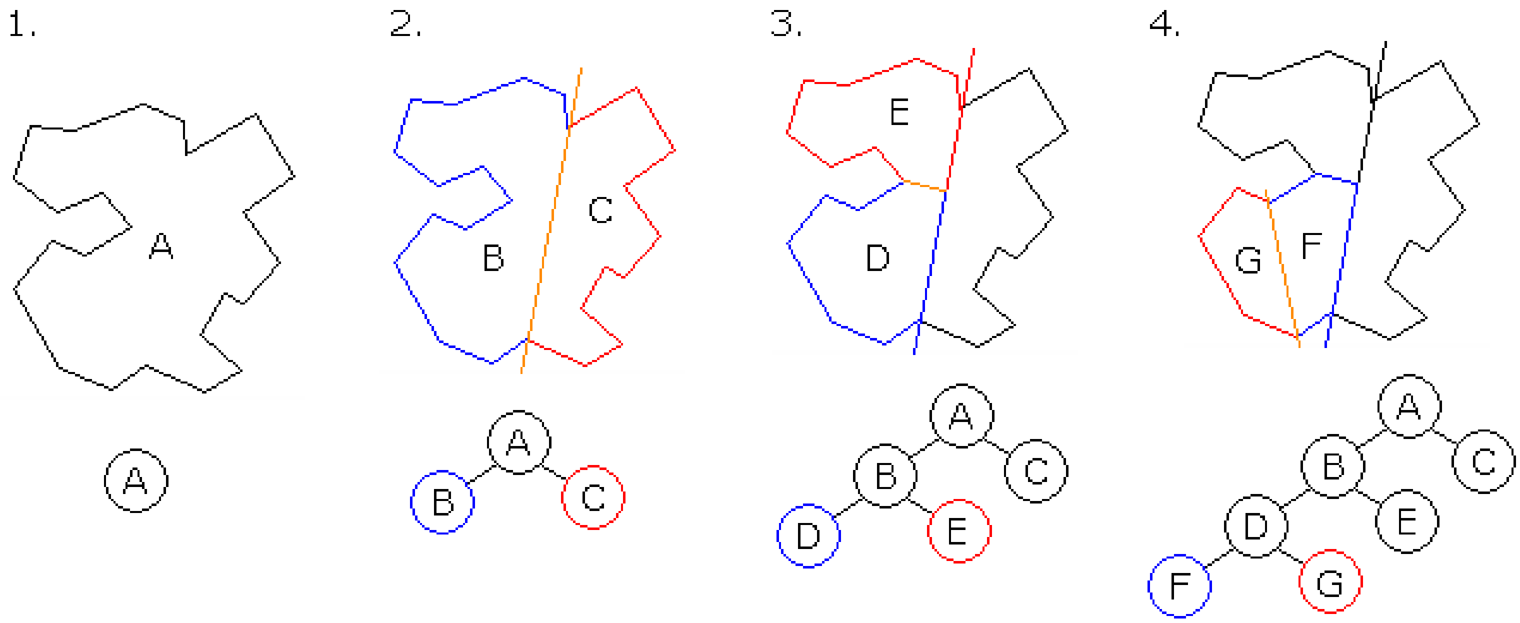






Binary Space Partition

Recursively split space using planes

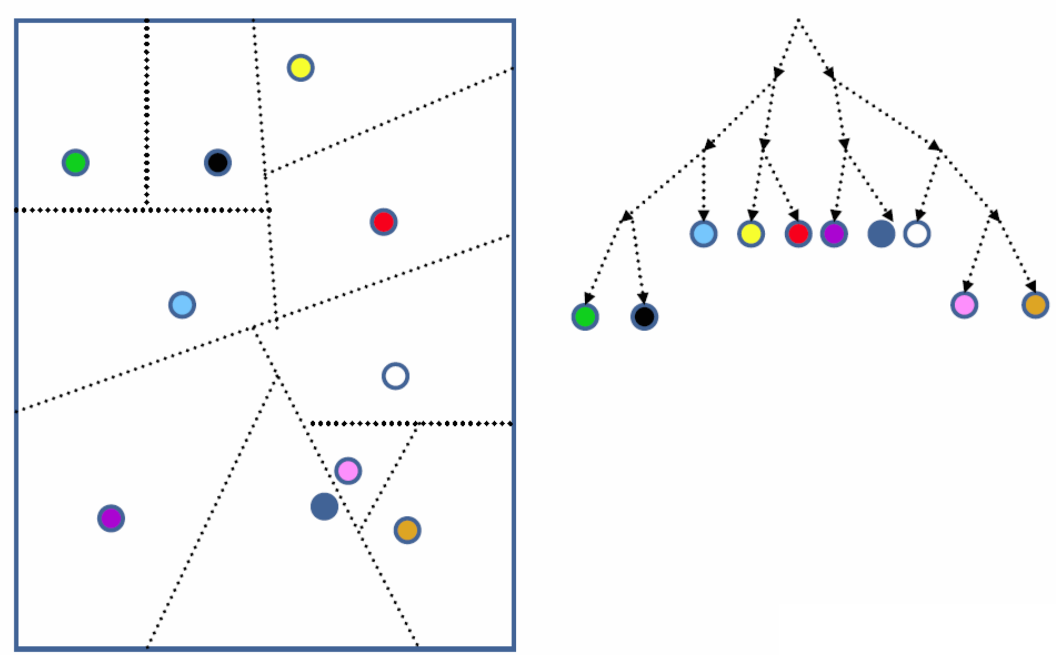


Binary Space Partition

Recursively split space using planes

Each node stores splitting plane

Each leaf stores object references



Binary Space Partition

Recursively split space using planes

Each node stores splitting plane

Each leaf stores object references

How to pick good splitting plane?

Binary Space Partition

Recursively split space using planes

Each node stores splitting plane

Each leaf stores object references

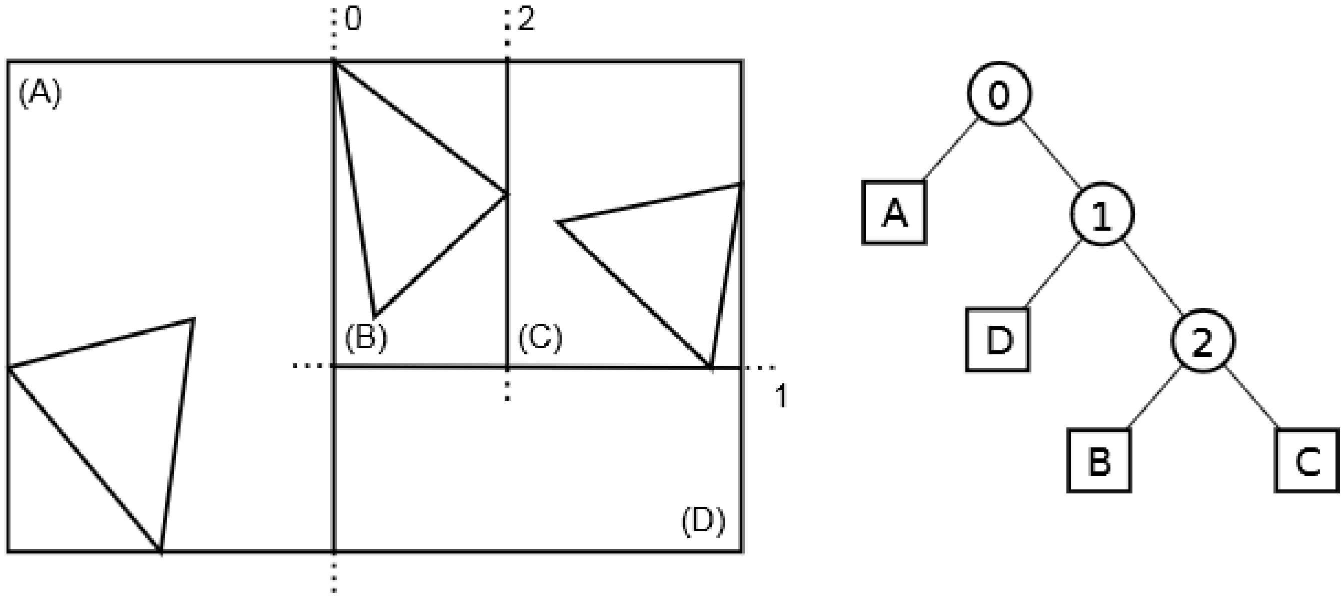
How to pick good splitting plane?

- heuristics / black magic
  - good partitioning vs good balance
- special case: axis-aligned planes

kD Tree

"k-Dimensional Tree"

BSP where each node is vertical or horizontal plane



kD Tree

How to pick splitting plane?

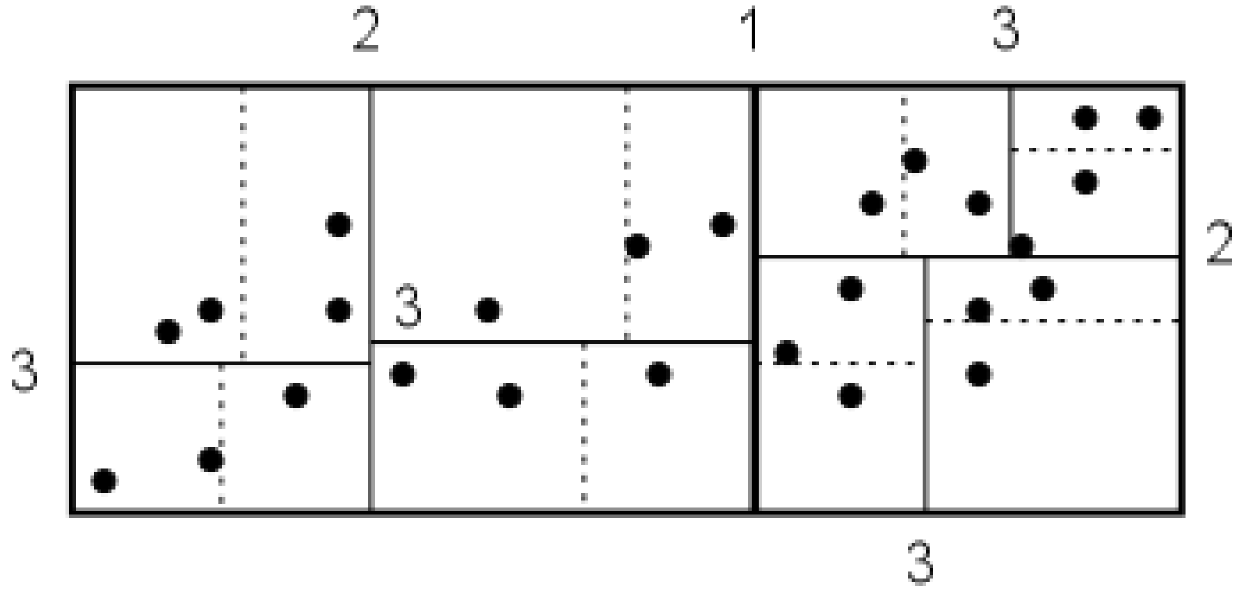
Goals:

- balance area of two children
- balance number of objects in children
- avoid **splitting** objects

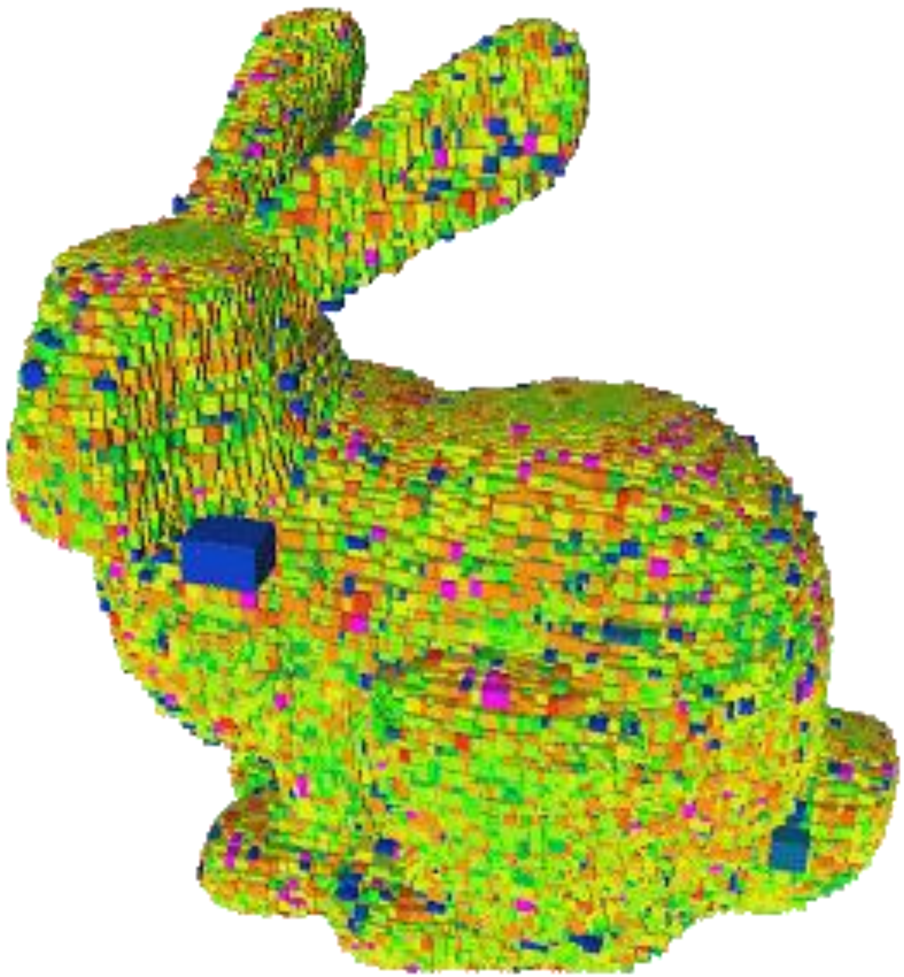
kD Tree

How to pick splitting plane?

Common strategy: split next to **median** object along **longest direction**



3D Tree



kD Tree

Pros:

- can tailor cell shape to fit objects
- balanced tree

Cons:

- cells not uniformly placed or shaped
- must pick good max tree depth

Devils Lurk in the Details

Building the leaves:

- what is the bounding box? (AABBs)
- is my object inside, outside, or crossing a grid cell? (spatial hash/octree)
- is my object on the left, right, or both sides of the split plane? (BSP/kD tree)
- how do I duplicate object references correctly? (all but BVHs)

Devils Lurk in the Details

Traversing the tree:

- how exactly do I do ray-node intersection?
  - ray/box (AABBs, octree)
  - ray/plane (BSP and kD trees)

Devils Lurk in the Details

Traversing the tree:

- how exactly do I do ray-node intersection?
  - how do I do it efficiently?

- what if my ray starts inside the scene?

Kinetic Data Structures

During animation, objects move slowly

Cumulatively **update** data structures instead of rebuilding every frame

Kinetic Data Structures

During animation, objects move slowly

Cumulatively **update** data structures instead of rebuilding every frame

Easy:

- spatial hash
- octree

Annoying:

- BSP trees (kD trees)