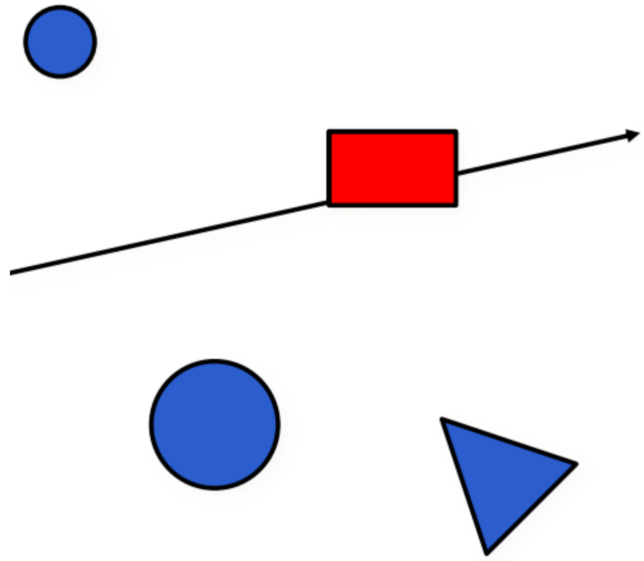
# Accelerating Data Structure

In ray tracing, we need to determine weather a ray intersects with an object. Naively by brute force this is done by

1. Intersect ray with every primitive
2. Take closest intersection



Brute-force intersection detection

> The brute-force method is extremely inefficient. We need more advanced techniques to make intersection detection more efficient, either by preprocessing the scene or make some change to the ray.

## Acceleration Technique Overview

- 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

We will mainly focus on "Fewer intersection computations" in this article.

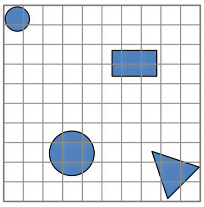
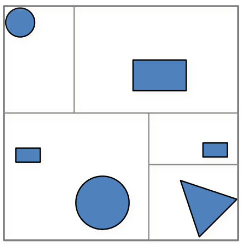
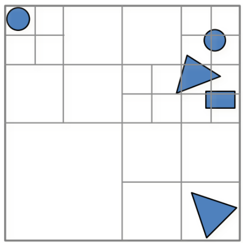
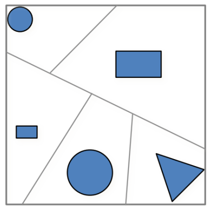
## Spatial Decomposition

Generally spatial decomposition techniques are composed of two steps

1. Preprocess
   1. Decompose space into disjoint regions
   2. Store pointers to overlapping objects within each region
2. Rendering

2. Traverse through regions overlapping the ray
3. Intersect objects in each region until a hit is found

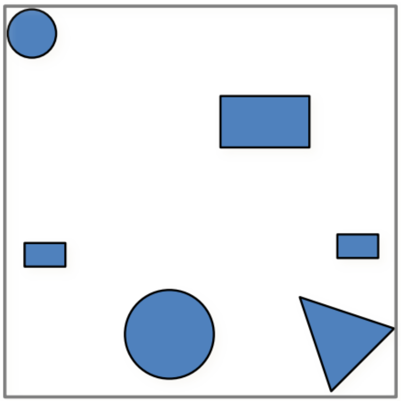
We will see 4 spatial decomposition methods as shoed below

| **Uniform** | **KD-tree** | **Octree** | **BSP-tree** |
|---|---|---|---|
|  |  |  |  |
| Built at once | Fixed plane orientation, variable position and axis | Fixed splitting operation | Arbitrary planes |

## Uniform Grid

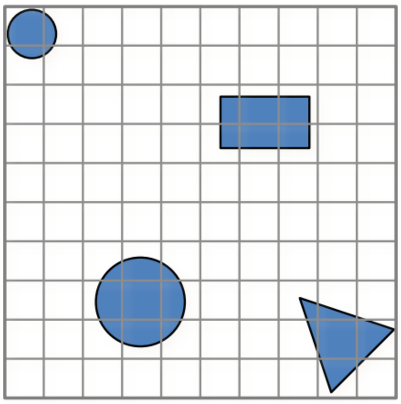
### Preprocessing

1. Compute bounding box (of the scene)



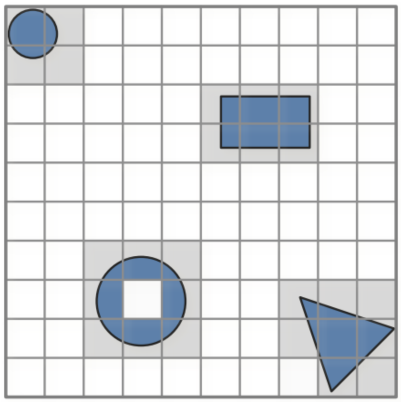
Bounding box of the scene

2. Determine grid resolution (often $\sim 3n^{1/3}$)



Bounding box in grid

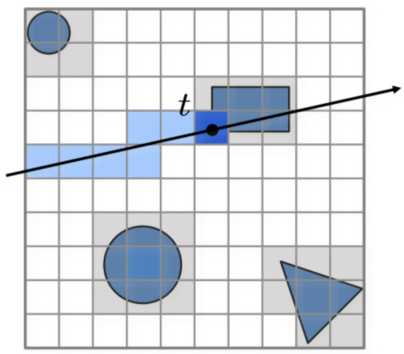
3. Insert objects into cells
4. Rasterize bounding box
5. Prune empty cells



Prune empty cells

6. Store reference for each object in cell

### Ray Intersection




How ray intersects with objects

1. Incrementally rasterize ray
2. Compute intersection with objects in each cell
3. Stop when intersection found in current voxel

**Pros**:

- Easy to code, building data structure is fast

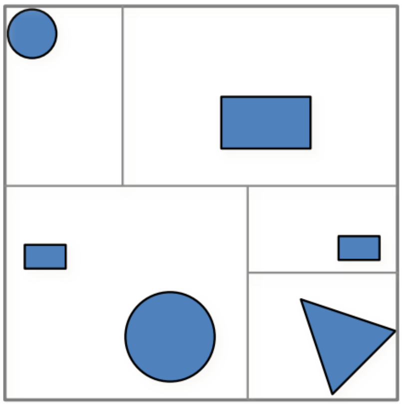
**Cons**:

- Uniform cells do not adapt to non uniform scenes

## KD-Trees (Wikipedia)

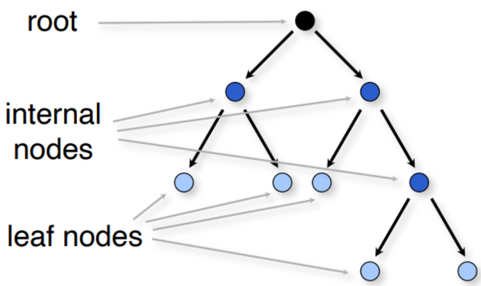
### Preprocessing

1. Compute bounding box of the scene
2. Recursively split cell using axis-aligned plane, until termination criteria, e.g. maximum depth or minimum number of objects attained



Space division by KD-tree

3. Build binary tree structure




Binary tree structure

    1. Internal nodes store
        1. Split axis: x, y or z axis
        2. Split position: coordinate of split plane along axis
        3. children: reference to child nodes
    2. Leaf nodes store
        1. List of primitives
        2. Optionally: mailboxing information

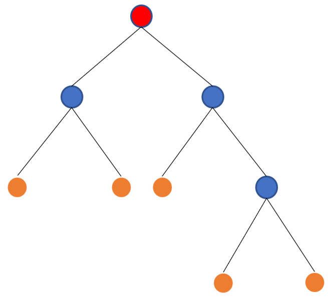
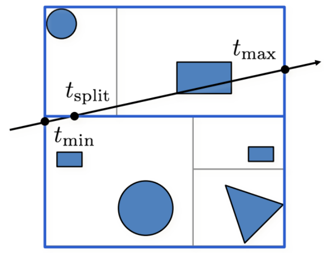
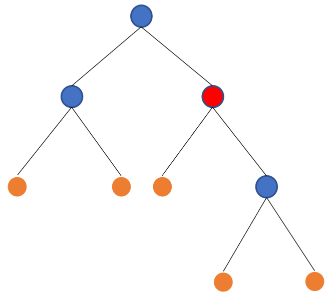
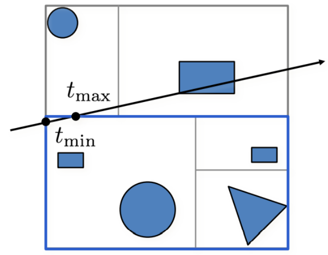
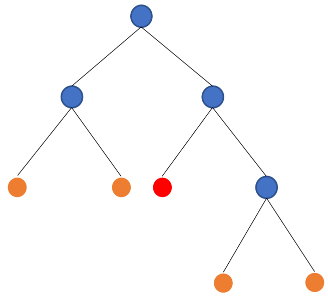
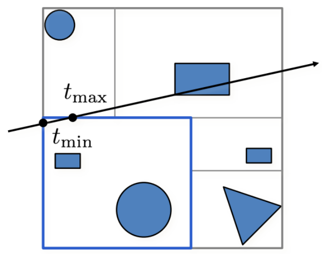
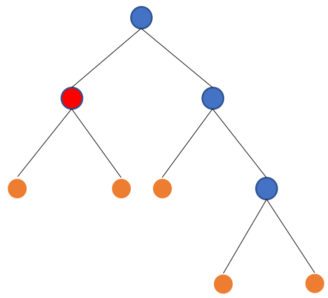
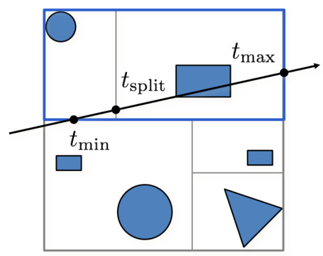
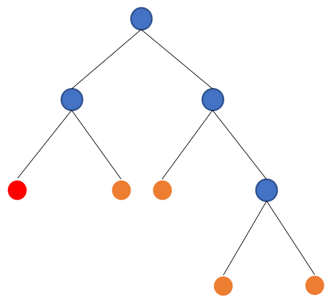
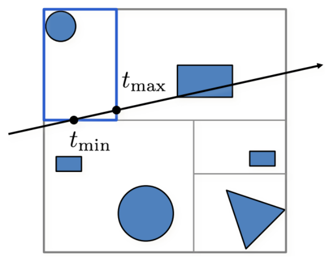
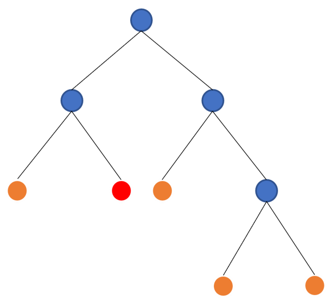
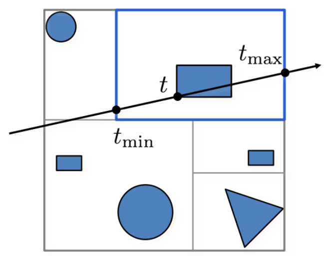
Since there are many possible ways to choose axis-aligned splitting planes, there are many different ways to construct KD trees, which are explained in details here. The canonical method of KD tree construction is:

- As one moves down the tree, one cycles through the axes used to select the splitting planes. (For example, in a 3-dimensional tree, the root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have z-aligned planes, the root's great-grandchildren would all have x-aligned planes, the root's great-great-grandchildren would all have y-aligned planes, and so on.)
- Points are inserted by selecting the median of the points being put into the subtree, with respect to their coordinates in the axis being used to create the splitting plane. (Note the assumption that we feed the entire set of $n$ points into the algorithm up-front.)

## Ray Intersection

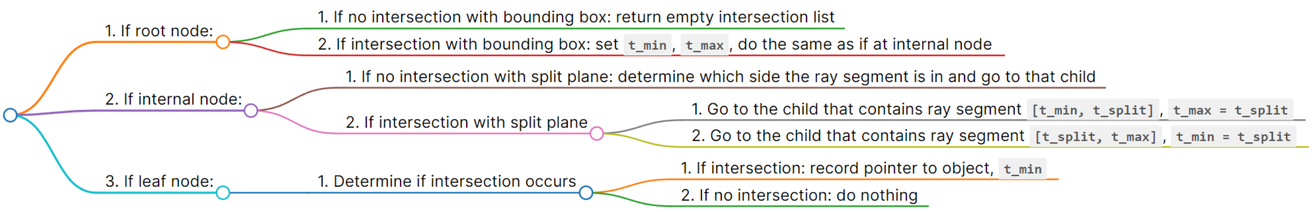
The intersection is done by a top-down recursion

| Current node | Current box |
| --- | --- |
| | |

<table>
<tr><th>Current node</th><th>Current box</th></tr>
<tr><td></td><td>
</td></tr>
<tr><td></td><td>
</td></tr>
<tr><td></td><td>
</td></tr>
<tr><td></td><td>
</td></tr>
<tr><td></td><td>
</td></tr>
<tr><td></td><td>
</td></tr>
</table>

1. If root node:
    1. If no intersection with bounding box: return empty intersection list

    2. If intersection with bounding box: set $t_{min}$, $t_{max}$, do the same as if at internal node
2. If internal node:
    1. If no intersection with split plane: determine which side the ray segment is in and go to that child
    2. If intersection with split plane
        1. Go to the child that contains ray segment $[t_{min}, t_{split}]$, $t_{max} = t_{split}$
        2. Go to the child that contains ray segment $[t_{split}, t_{max}]$, $t_{min} = t_{split}$
3. If leaf node:
    1. Determine if intersection occurs
        1. If intersection: record pointer to object, $t_{min}$
        2. If no intersection: do nothing



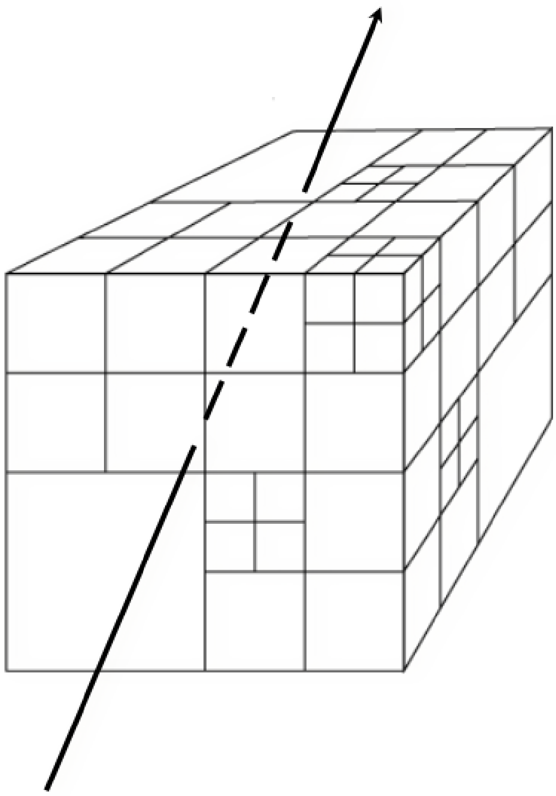

## Octrees

### Preprocessing

1. Compute bounding box
2. Recursively subdivide cells into 8(4 for 2D space) equal sub-cells until termination criteria attained

### Ray Intersection

Similar to KD trees



**Pros**:

- Easier to implement
- Cheaper costs for insertion and deletion

**Cons**:

- Generally less effective division of space

### General BSP(Binary Space Partition) trees

#### Preprocessing

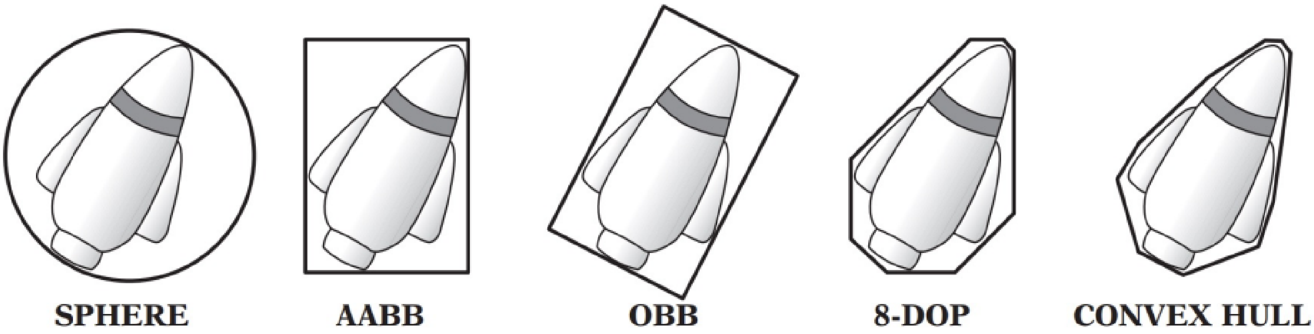
1. Compute bounding box
2. Recursively split space using arbitrary planes until termination criteria attained

## Object Decomposition

Decompose objects into (overlapping) sets & bound using simple volumes for fast rejection.

### Bounding Volume Hierarchies

6 kinds of bounding volume are listed here



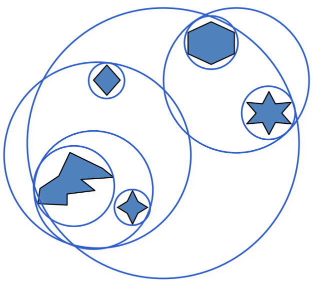
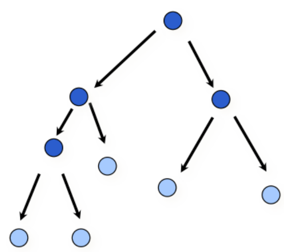
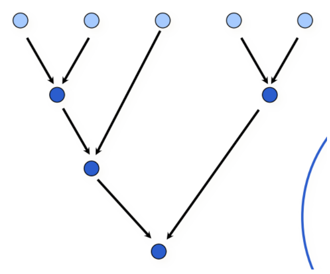

with:

1. AABB for Axis-Aligned Bounding Box
2. OBB for Oriented Bounding Box
3. k-DOP for k-Discrete Orientation Polytopes

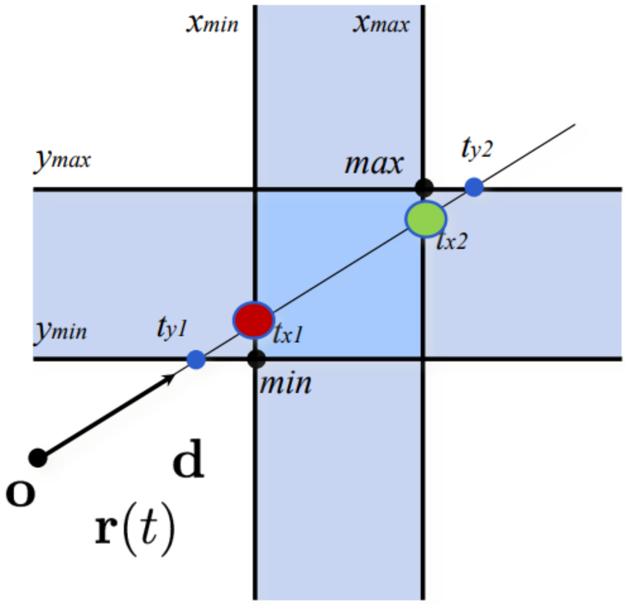
The tradeoff between different bounding volumes is:

- complex shape → tight fit → fewer intersections
- simple shape → fast intersection & less memory

The bounding volume hierarchies could be constructed either top down or bottom up

| | Top down | Bottom up |
| --- | --- | --- |
|  |  |  |

#### Ray-AABB Intersection




Ray-AABB intersection algorithm

$$
\begin{aligned}
\mathbf{o}_x + t_{x_1}\mathbf{d}_x &= x_{min} \\
\mathbf{o}_x + t_{x_2}\mathbf{d}_x &= x_{max}
\end{aligned}
$$

> **Algorithm**:
>
> 1. Solve for $t_{x_1}$ and $t_{x_2}$
>
> $$t_{x_1} = \frac{x_{min} - \mathbf{o}_x}{\mathbf{d}_x}, \quad t_{x_2} = \frac{x_{max} - \mathbf{o}_x}{\mathbf{d}_x}$$
>
> 2. If $t_{x_1} > t_{x_2}$, swap($t_{x_1}$, $t_{x_2}$)
> 3. Repeat for $t_{y_1}$, $t_{y_2}$, $t_{z_1}$, $t_{z_2}$.
> 4. Set $t_{min}$ and $t_{max}$
>
> $$
> \begin{aligned}
> t_{min} &= \max(t_{x_1}, t_{y_1}, t_{z_1}) \\
> t_{max} &= \min(t_{x_2}, t_{y_2}, t_{z_2})
> \end{aligned}
> $$
>
> 5. Hit if $t_{min} < t_{max}$