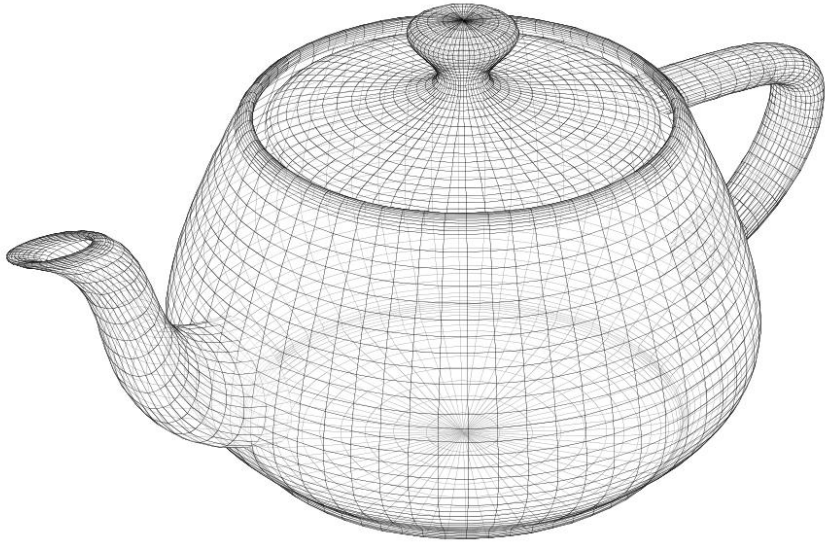


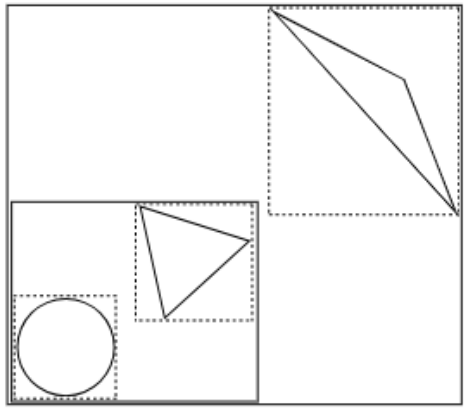
Geometric Data Structures

Bounding Volume Hierarchies (BVHs)

Production Computer Graphics
Eric Shaffer

ILLINOIS

Bounding Volume Hierarchies (BVHs)



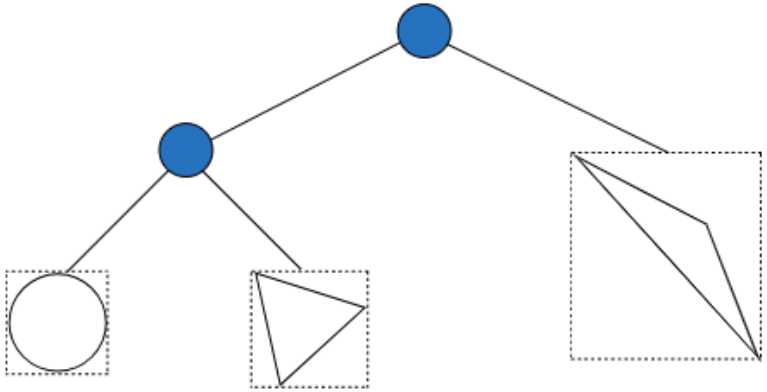
(a)

BVHs uses primitive (ie geometric objects) subdivision

- The set of primitives is partitioned into a hierarchy of disjoint sets
- Unlike grids or kd-trees which partition space

(a) Shows a set of primitives

- Bounding boxes shown by dashed lines
- The primitives are aggregated based on proximity



(b)

(b) Shows he corresponding bounding volume hierarchy.

- The root node holds the bounds of the entire scene.
- Primitives are found at the leaves

ILLINOIS

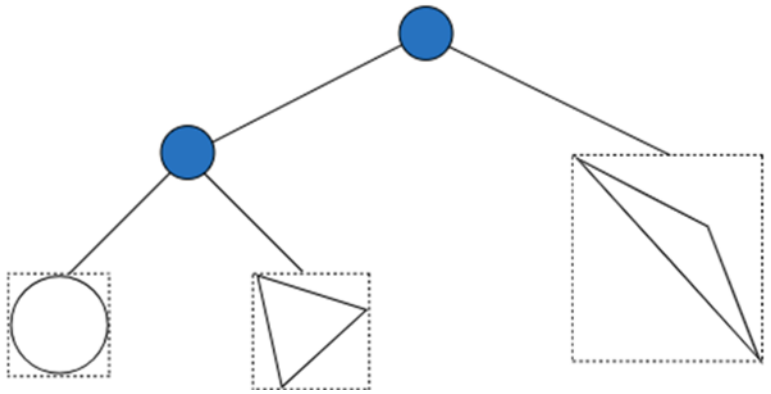
Storage Properties of BVHs

Each primitive appears in the hierarchy only once

- In spatial partition, an object can overlap multiple partitions

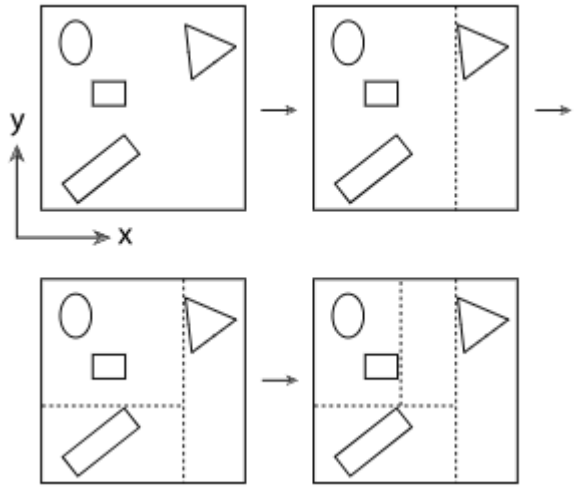
Memory needed for a BVH is bounded

- Assuming a binary tree N objects
- N leaf nodes
- N-1 interior nodes

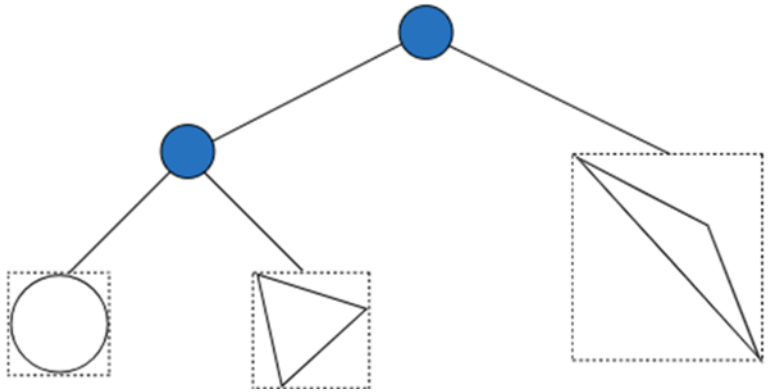


BVH vs kd-tree

- BVH is much more efficient to build
- Kd-tree has somewhat faster ray intersection tests with nodes
- BVHs more numerically robust
  - Less prone to missed intersection due to round-off error



Kd-tree construction



BVH

```
#ifndef BVH_H
#define BVH_H

#include "rtweekend.h"

#include "hittable.h"
#include "hittable_list.h"


class bvh_node : public hittable {
    public:
        bvh_node();

        bvh_node(const hittable_list& list, double time0, double time1)
            : bvh_node(list.objects, 0, list.objects.size(), time0, time1)
        {}

        bvh_node(
            const std::vector<shared_ptr<hittable>>& src_objects,
            size_t start, size_t end, double time0, double time1);

        virtual bool hit(
            const ray& r, double t_min, double t_max, hit_record& rec) const override;

        virtual bool bounding_box(double time0, double time1, aabb& output_box) const override;

    public:
        shared_ptr<hittable> left;
        shared_ptr<hittable> right;
        aabb box;
};

bool bvh_node::bounding_box(double time0, double time1, aabb& output_box) const {
    output_box = box;
    return true;
}

#endif
```

A Specific Example

From *Ray Tracing the Next Week* by Peter Shirley

ILLINOIS

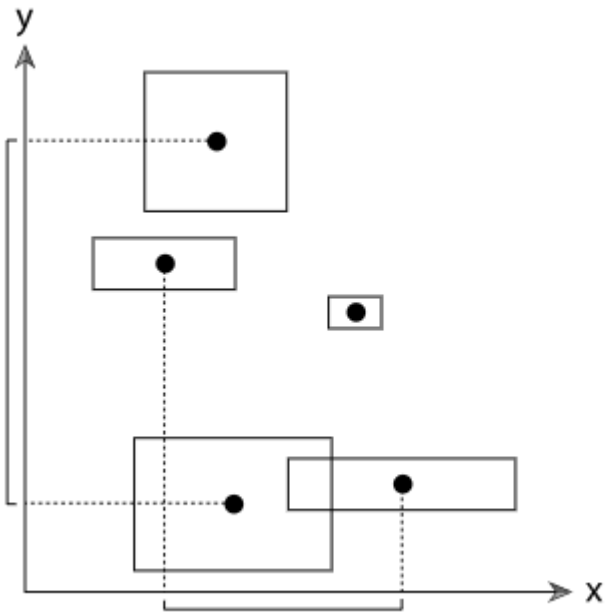
Partitioning Heuristics

Must partition objects into 2 groups

- $2^n - 2$ possible non-empty binary partitions of n objects

Simplify by choosing a partition along one of the three coordinate axes

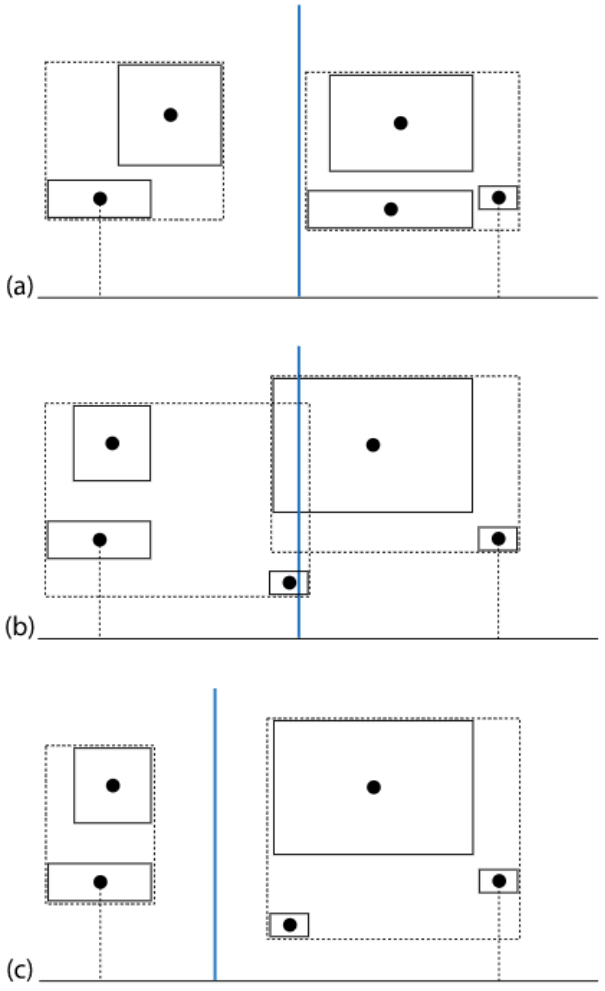
- Approximately $3n$ candidate partitions



- Compute centroid for each bounding box
- Pick the axis to partition based on largest range of centroid projections:
- Compute $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$ for centroids
- Pick axis based on largest of $x_{max} - x_{min}$, $y_{max} - y_{min}$, $z_{max} - z_{min}$
- In example on left, we partition (ie cut) the y axis

ILLINOIS

Where to Partition: Midpoint Method



If we are splitting on x-axis, split at $\frac{x_{max}-x_{min}}{2}$

(a) Shows a midpoint split

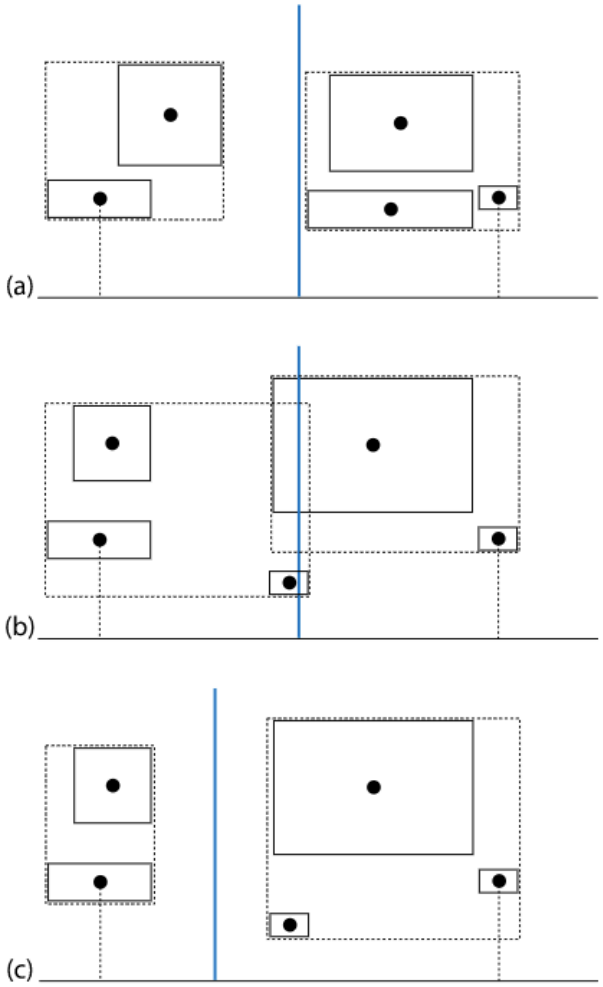
(b) Shows a midpoint split that is sub-optimal

(c) Shows a better (non-midpoint) split

Overlapping AABBs means both branches of tree may need to be traversed during ray intersection…not ideal for performance

ILLINOIS

Where to Partition: Median Method



If we are splitting on x-axis, split into 2 groups

- One contains objects with $\frac{n}{2}$ smallest centroids by x coordinate
- Other contains objects with $\frac{n}{2}$ largest centroids by x coordinate

This can find a partition like (c)

Can be done in $O(n)$ time

> This scheme is also easily implemented with a standard library call, `std::nth_element()`. It takes a start, middle, and ending pointer as well as a comparison function. It orders the array so that the element at the middle pointer is the one that would be there if the array was fully sorted, and such that all of the elements before the middle one compare to less than the middle element and all of the elements after it compare to greater than it. This ordering can be done in $O(n)$ time, with n the number of elements, which is more efficient than the $O(n \log n)$ of completely sorting the array.

Physically Based Rendering by Pharr et al

ILLINOIS

Where to Partition: Surface Area Heuristic

SAH attempts to find partition optimal-ish for ray intersection tests

When Constructing Tree

- At any time, we could make the current region a leaf node
- Any ray passing through region will be tested against all objects in it
- Computational cost will be $\sum_{i=1}^{N} t_{\text{isect}}(i)$

N is number of primitives and $t_{isect}(i)$ is the time to compute a ray–object intersection with the ith primitive.

SAH

Other option is to split region

In this case a ray crossing the region has computational cost

$$c(A,B) = t_{\text{trav}} + p_A \sum_{i=1}^{N_A} t_{\text{isect}}(a_i) + p_B \sum_{i=1}^{N_B} t_{\text{isect}}(b_i)$$

t_{trav} is time to which child node the ray will traverse

P_A is probably of ray hitting child A

P_B is probably of ray hitting child B

ILLINOIS

Computing SAH

Common to assume $t_{isect}(i)$ is same for all objects

Compute the probabilities using $P(B|A) = \frac{S_B}{S_A}$

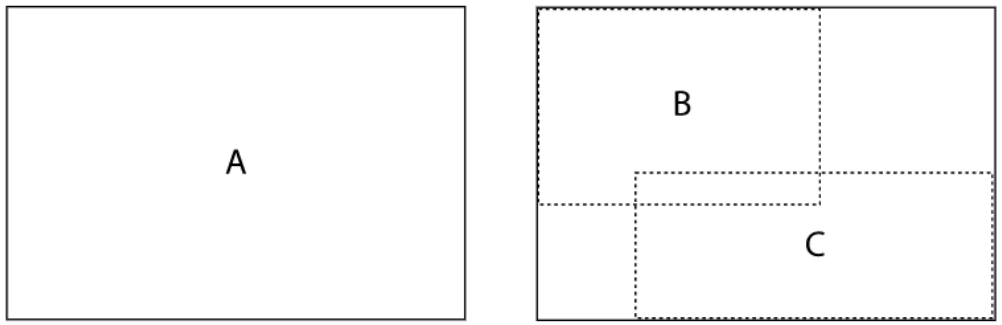
Physically Based Rendering
by Pharr et all
uses intersection cost of 1
and traversal cost of 1/8

A is the parent node bounding box (or whatever convex volume is being used)

B is the child node bounding box

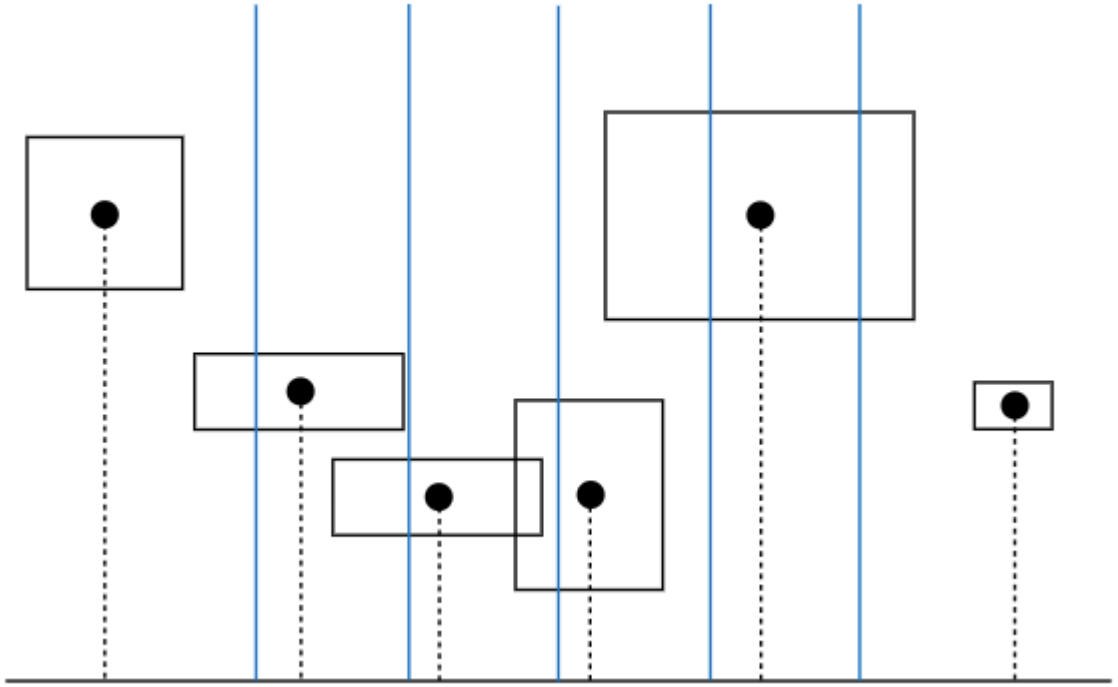
S_A is the surface area of region A

S_B is the surface area of region B



ILLINOIS

Computing SAH



To Choose Partition

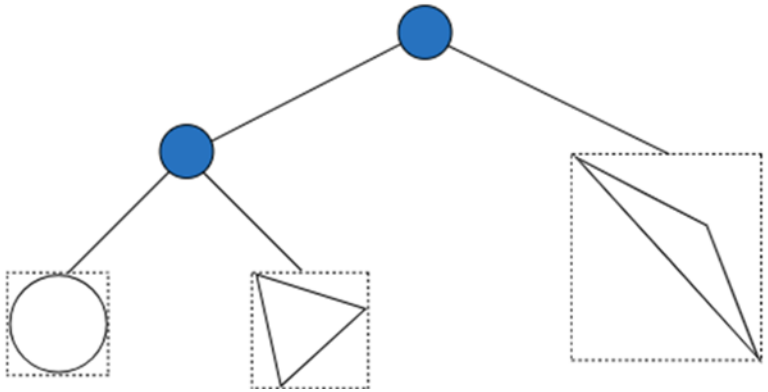
- Each centroid is considered to be in a bucket
- Compute SAH for each partition based on a bucket boundary
  - Bucket boundaries are the blue lines
- Choose minimum SAH cost partition

Process takes $O(n^2)$ time

$$c(A,B) = t_{\text{trav}} + p_A \sum_{i=1}^{N_A} t_{\text{isect}}(a_i) + p_B \sum_{i=1}^{N_B} t_{\text{isect}}(b_i)$$

ILLINOIS

Traversal



Simple…

- Test against root
- If ray enters root bounding volume
- Test against each child BV
- Recurse (or iterate) until you reach a leaf