Acceleration Structures

George Tang

May 15, 2019

1 Introduction

Recall for rasterization, we discussed an optimization when for filling the triangle, we only search in the bounding box of the projection of the triangle. For raytracing, because each pixel's value is calculated by testing the ray associated with it against the scene objects, we can impose a partitioning scheme among the objects to accelerate the search process. The simplest scheme is to draw a bounding box around the object. We test all rays that intersect the box with the object (e.g. a teapot composed of hundreds of triangles); otherwise don't. Clearly, this will greatly accelerate the rendering process. This process is implemented through **acceleration structures**, data structures that support partitioning schemes.

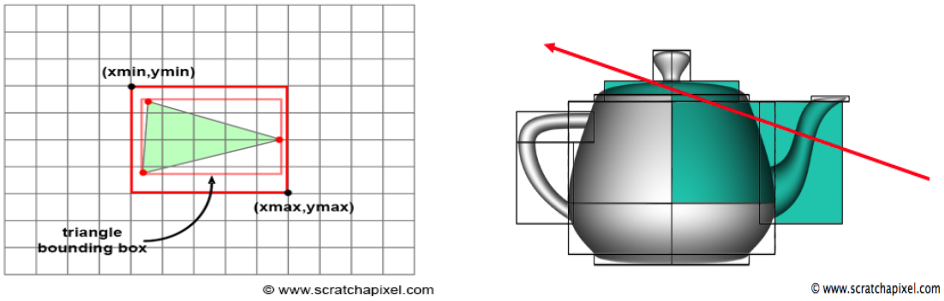


Figure 1: 2D Bounding Boxes for Rasterization (left) vs. 3D Bounding Boxes for Raytracing (right)

2 Ray-Box Intersection

The simplest type of bounding box to deal with is the **axis aligned bounding box (AABB)**. As its name suggests, its sides are parallel to the x,y, z axis. For each bounding box we define vmn and vmx, which are two diagonally opposite vertices, representing the least and greatest extent of the AABB from the perspective of the eye.

A ray will have at most two intersection points with a finite box. If we extend the sides of the AABB infinitely, we will always have two intersections. Determining the distances to these intersections for the infinite AABB will allow us to determine if the ray intersects the definite AABB. We first only consider the case where there are no z bounding planes (x and y bounding planes extend infinitely).

again, our line can be defined parametrically and distance between the line and the lower x plane can we derived from

$$\mathbf{O}.x + t_{mnx}\mathbf{d}.x = vmn.x$$
$$t_{mnx} = \frac{(vmn.x - \mathbf{O}.x)}{\mathbf{d}.x}$$

And t_{mxx}, t_{mny}, and t_{mxy} can be found using the same process.

We have two cases, illustrated in the diagram above. To test for the miss case, we test if $t_{mnx} > t_{mxy}$ or $t_{mny} > t_{mxx}$. We do the same for x and z bounding planes only. We have an intersection only if both x and

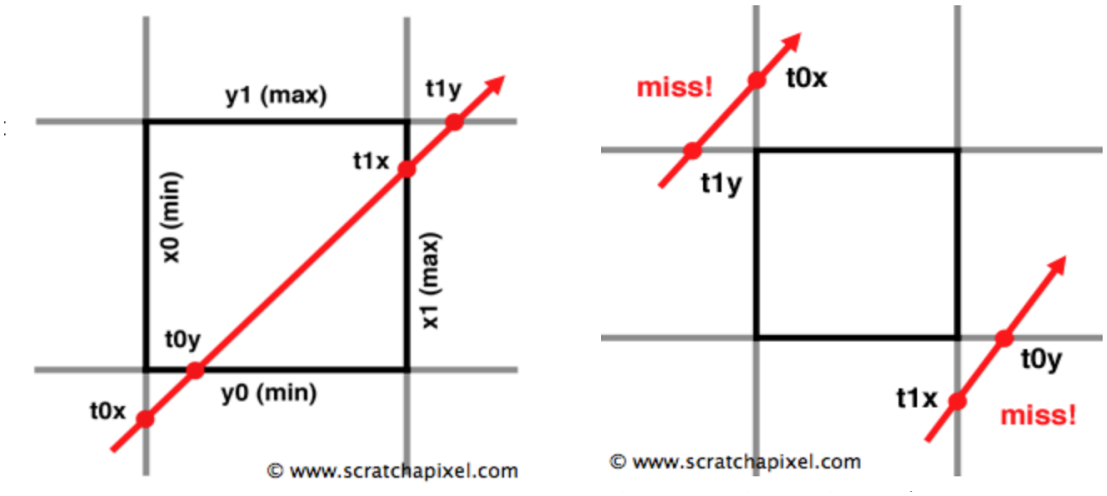
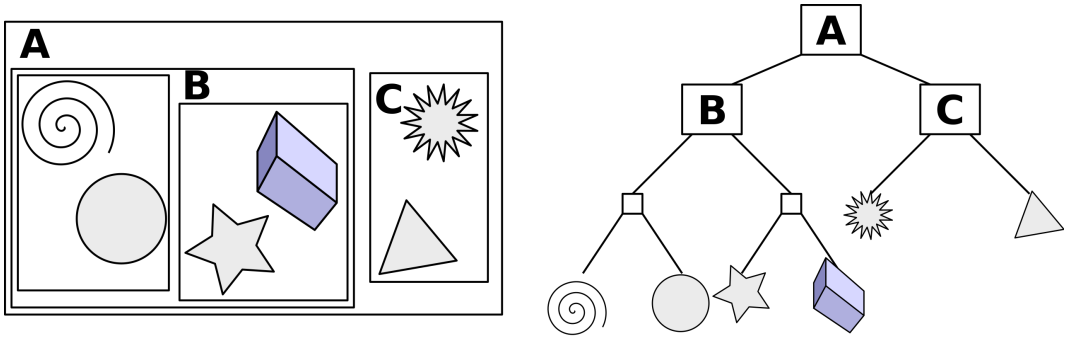


Figure 2: Hit case vs. Miss Case

y and x and z hit. A optimized implementation can be found at https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection.

3 Bounding Volume Hierarchy

Note if we have groups of individual volumes, we can group them and so forth. We can establish a hierarchy among the groups, known as a **Bounding Volume Hierarchy (BVH)**, further accelerating computation.



4 KD-Tree

One way one the best BVHs (speed-wise and memory-wise) is the KD-Tree data structure. A KD-Tree consists of a build and a traversal algorithm. We begin with the root node encompassing all objects. Naive KD-build recursively assigns each object in a node to the left child or right child or both based on the object's location relative to the node's axis-aligned splitting plane. This plane is determined to be located at the median of the axis components of the triangles' centroids (represented as vectors). As we move down the tree during construction, we cycle through the x, y, z-axes and repeat. If the number of triangles is less than a certain threshold, then we stop splitting and denote the node as a leaf. Note only leaf nodes contain triangles.

In the end, the axis-aligned splitting planes form bounding boxes that are used to determine if the ray intersects with any triangle in the group of triangles contained by the box. Because a KD-Tree divides the search space in two each time, the maximum depth is $log(N)$. Because we sort the triangles by axis component to determine the splitting plane when we construct each node, the overall time complexity for KD-build is $O(Nlog^2(N))$

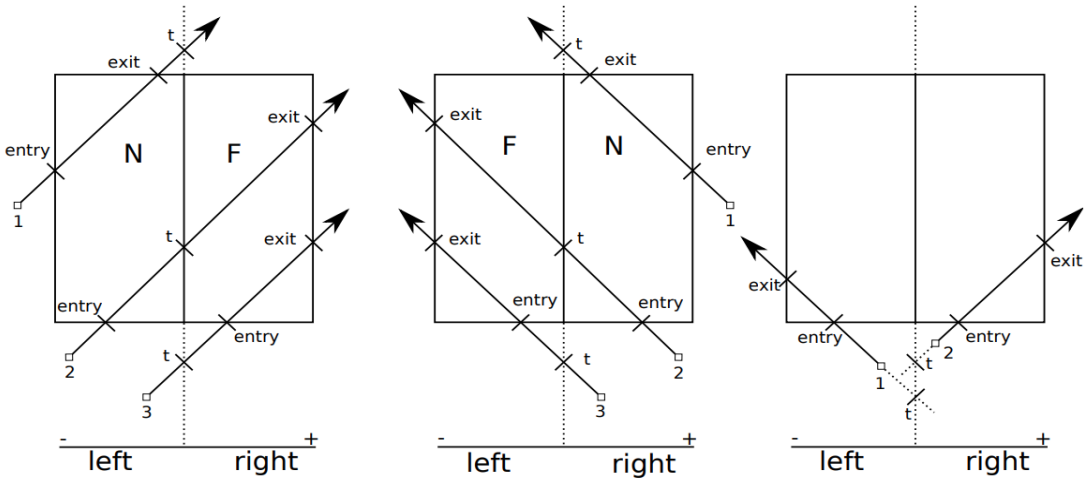


Figure 3: Casework for KD-traverse (see Algorithm 2). t refers the ray's distance to the splitting plane. If t is greater than the exit distance, then it only intersects the near node. If t is less, then it only intersects the far node. If t is in between, we have to search both nodes (from [**?**]).

Algorithm 1 Naive KD-Build with SAH

```
function FindPlaneSAH(V, axis)
    best = none
    minc = INF
    for cent[axis] in T do
        V_L, V_R ←Split(cent[axis])
        curr = Cost(V_L, V_R)
        if curr < minc then
            minc ← curr
            best ← cent[axis]
    return best
function KDBuild(V, depth)
    if Terminate(V) then
        return
    axis ← depth % 3
    p ←FindPlaneSAH(V, axis)
    V_L, V_R ←Split(p)
    T_L ← {t ∈ T | t ∩ V_L ≠ ∅}
    T_R ← {t ∈ T | t ∩ V_R ≠ ∅}
    KDBuild(V_L)
    KDBuild(V_R)
```

Algorithm 2 Naive KD-Traverse

```
function KDTraverse(V_root, ray)
    stack.push(V_root)
    while stack not empty do
        (V, ent, ext) ← stack.pop
        while V not leaf do
            t ← DisToPlane(V.p)
            V_near, V_far ←Classify(V_L, V_R)
            if t ≥ exit or t < 0 then
                V ← V_near
                continue
            if t ≤ ent then
                V ← V_far
            else
                stack.push(far, t, ext)
                V ← V_near
                ext ← t
        if FindIntersection(V, ray) then
            return intersection
    return ∅
```

The traversal algorithm is also recursive. Starting with the root node, we determine if the ray intersects it. If so, then we determine if it intersects only the left child, only the right child, or both, and traverse in accordance (Figure 2). If we reach a leaf node, we test intersection with every triangle in the node. The time complexity for KD-traverse is $O(log(N))$.

One optimization we can make is instead of determining the splitting value based on the median of the axis component of the centroid, we locally maximize the probability that the ray will have to search fewer triangles. Intuitively, this translates to maximizing empty space in the bounding boxes, and formally we define the Surface Area Heuristic, which has been proven to be robust in almost all conditions.

The cost to traverse a node can be estimated given a splitting configuration, and we want to select the min cost configuration:

$$Cost(V_L, V_R) \approx K_T + K_I \left(\frac{SA(V_L)}{SA(V)} T_L + \frac{SA(V_R)}{SA(V)} T_R \right)$$

where V_L, V_R, V are the left, right, and parent voxels (nodes), K_T is the cost of a bounding box intersection, K_I, the cost of a triangle intersection, $SA(V_L)$, $SA(V_R)$, $SA(V)$, the surface area of the left, right, and parent nodes, and T_L, T_R the number of triangles in the left and right nodes.

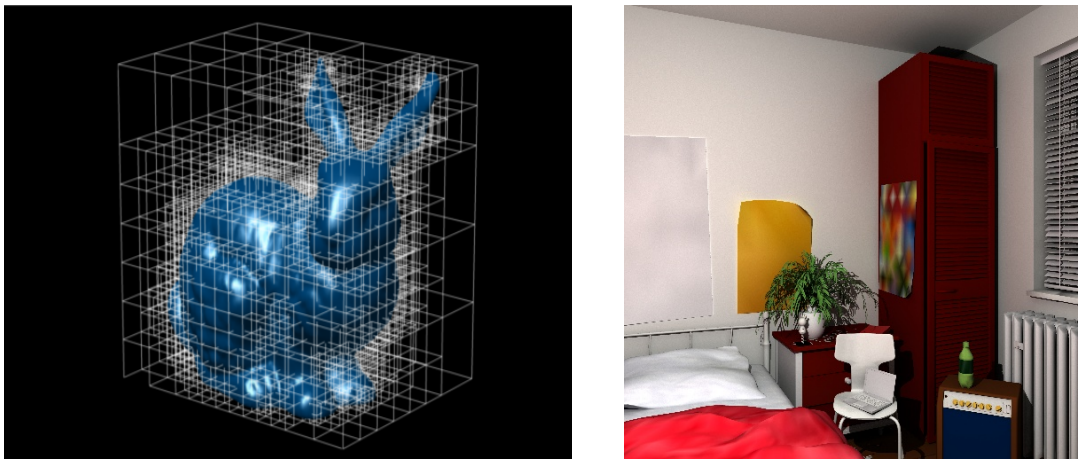


Figure 4: KD-Tree Partitioning of Stanford Bunny Model (left) and A Personal Rendering using KD-Tree (right)