Acceleration Structures for Ray Tracing: k-d Trees and Bounding Volume Hierarchies

Gary Gurlaskie

CAP5705 Computer Graphics, Dr. Entezari

1 Project Goals

The goal of this project is to improve the efficiency of the ray tracer by using the bounding volume hierarchy (BVH) and k-d tree acceleration structures. In the project, I will implement both of these structures, and use them individually to improve the ray-hit efficiency of the ray tracer in order to render more complicated scenes interactively. I also will add an OBJ file reader to my ray tracer. Finally, I will compare the effectiveness of BVHs and kd-trees at improving ray tracing performance.

2 Work

2.1 Code Changes

I implemented a k-d tree and BVH and integrated them into my ray tracer. Listed below is the interface for the BVH (the k-d tree is similar).

```
class BVH: public Intersectable {
public:
    static BVH* ConstructBVH(std::vector<Shape*>&);

    inline Intersection* intersect(Ray& r) { return BVH::Intersect(r, root, 0); }
    inline Intersection* intersect(Ray& r, Shape* ignore) { return BVH::Intersect(r, root, ignore); }

private:
    static Intersection* Intersect(Ray&, BVHNode*, int);
    static Intersection* Intersect(Ray&, BVHNode*, Shape*);
};
```

Listing 1: The interface for the BVH

The BVH and k-d tree implementations contain multiple construction algorithms: spatial median, object median, object mean, and surface area heuristic. These are selected at compile-time.

I adapted the GLM library into the ray tracer program to read OBJ files, and I added interactive controls to the ray tracer window.

2.2 Performance Testing

I performed comparative performance testing by rendering 3 different models found in the "smooth" package: "eagle", "castle", and "galleon" using the ray tracer. I measured rendering time as well as mean leaf depth and mean triangles per leaf. These metrics should give a good idea of both the tree quality/performance.

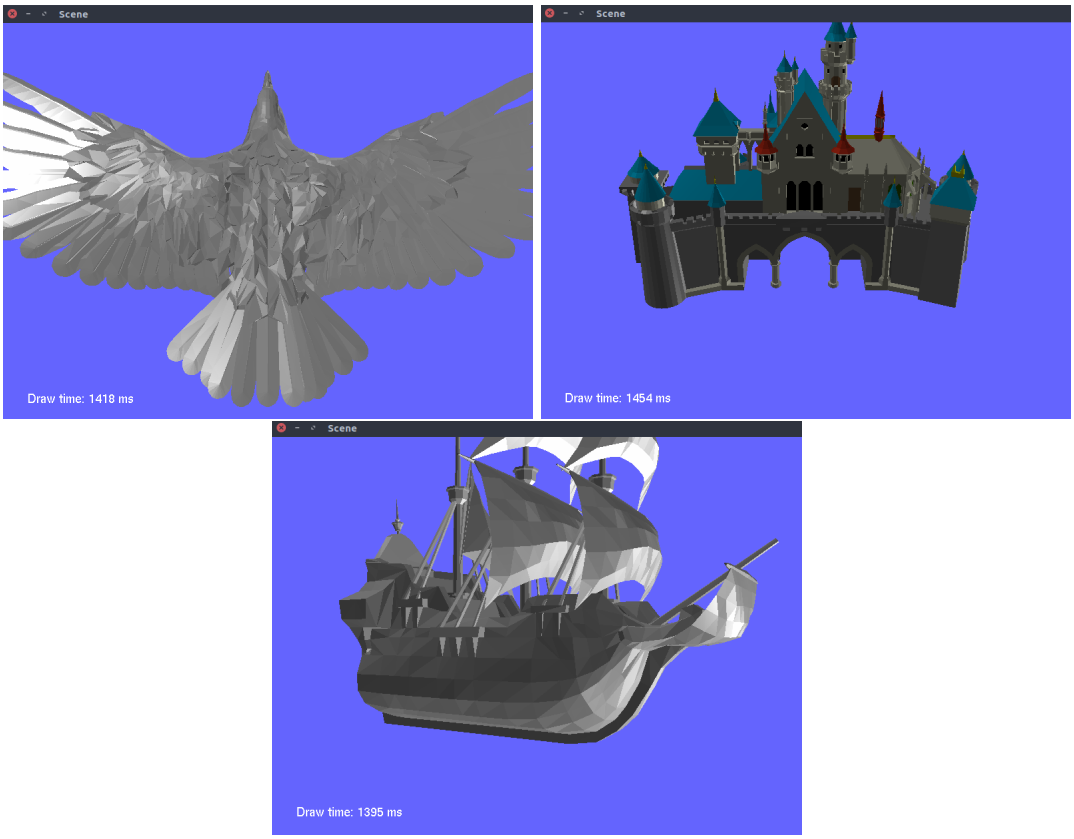

Figure 1: The "eagle", "castle", and "galleon" models used for testing. The models have 17327, 13114, and 4698 triangles respectively.

2.2.1 Tree Construction Speed

The object mean and spatial median algorithms constructed BVHs and k-d trees much faster than the Median and SAH algorithms. My implementation of the SAH was quadratic in the number of triangles, but this efficiency can be improved to log-linear. Below is the raw data.

Table 1: The tree construction speed.

Model	# Triangles	BVH-Spatial	BVH-Median	BVH-Mean	KD-Spatial	KD-Median	KD-Mean	KD-SAH
galleon	4k	5ms	79ms	10ms	22ms	206ms	28ms	36060ms
castle	13k	17ms	269ms	27ms	61ms	643ms	85ms	251319ms
eagle	17k	37ms	406ms	8ms	54ms	811ms	83ms	459217ms

2.2.2 Tree Quality

I used two measures of tree quality: mean triangles per leaf and mean depth per leaf. The mean triangles/leaf measure is especially relevant for k-d trees, since the same triangle can be placed in multiple subtrees. Below is the raw data. Notice that the object mean and object median perform the best in these measures.

Table 2: The mean leaf depth.

Model	# Triangles	BVH-Spatial	BVH-Median	BVH-Mean	KD-Spatial	KD-Median	KD-Mean	KD-SAH
galleon	4k	11	11	11	15	15	15	16
castle	13k	13	13	13	18	16	16	18
eagle	17k	16	13	13	17	16	16	18

Table 3: The mean triangles per leaf. This is not a very good metric for BVHs (since they directly partition the objects), but it is a good indicator of how well a k-d tree partitions the triangles.

Model	# Triangles	BVH-Spatial	BVH-Median	BVH-Mean	KD-Spatial	KD-Median	KD-Mean	KD-SAH
galleon	4k	5	2	1	19	12	12	14
castle	13k	9	2	1	23	10	10	16
eagle	17k	2	2	1	47	11	9	24

2.2.3 Rendering Time

The BVHs generally perform better than the k-d trees. The SAH k-d trees perform suprisingly poorly relative to the object-mean BVH, but this reflects the fact that the SAH-based partitioning can be made more efficient than the object-mean partitioning (which is always quadratic).

Table 4: The rendering performance.

Model	# Triangles	BVH-Spatial	BVH-Median	BVH-Mean	KD-Spatial	KD-Median	KD-Mean	KD-SAH
galleon	4k	9906ms	2829ms	1395ms	13179ms	4588ms	3816ms	2911ms
castle	13k	16137ms	2461ms	1021ms	9964ms	3525ms	2232ms	2224ms
eagle	17k	3532ms	1992ms	1418ms	14054ms	5444ms	5827ms	3015ms

3 Learning Outcomes

The main objects of study were the k-d tree and BVH data structures. I also learned about efficient k-d tree building using the SAH (although I did not implement the efficient version of the

algorithm). I will give a brief summary of these topics, and answer a question I was asked during my presentation.

3.1 k-d trees

A k-d tree is a space-partitioning data structure, for any number k of dimensions. The k-d tree stores a number of objects, and the k-d tree is wrapped in an axis-aligned bounding volume (AABV). The main idea is that you recursively divide the bounding volume using axis-aligned planes, and store the objects in the leaf nodes whose bounding volumes they intersect. The main difficulty is *building a good k-d tree*, that is, one that efficiently partitions the objects so that a single AABV can be queried in logarithmic time.

3.1.1 Building high-quality k-d trees for ray tracing

K-d trees that store triangles have a major problem, that some triangles have to be stored in multiple leaf nodes, since they cannot be separated from their neighbors with AABVs. There are a number of heuristics available to provide optimal partitioning, but they all have many trade-offs.

3.1.2 Spatial median partitioning

The simplest and most naive approach is to split the AABVs exactly in half, along their longest dimension. This is fast, but, as I found, it yields poor performance for ray-tracing. This approach would perform optimally on a model that has evenly-distributed triangles. You can see this with the "soccerball" model, that the performance of the spatial median technique is nearly identical to the performance of more advanced techniques.

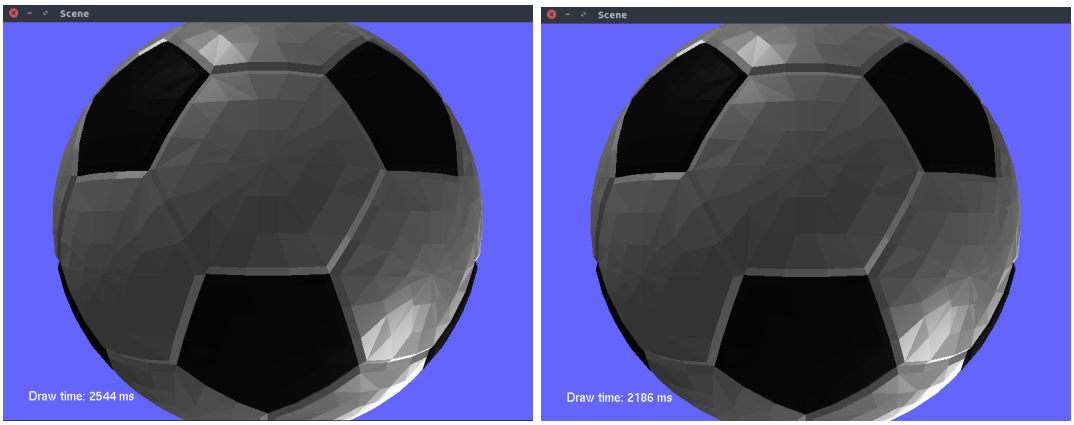

Figure 2: Left: "soccerball" rendered with spatial partitioning, Right: "soccerball" rendered with mean partitioning.

3.1.3 Object median partitioning

Another simple approach is to first sort the objects along the partitioning dimension, and create a split plane through the median object. This approach yields better performance on less-regular models, like "castle" or "galleon", which both have many "long" triangles (see Figure 1 and Table 4).

3.1.4 Object mean partitioning

This technique was not mentioned in [1], [2], or [3], but it yielded the best performance for both k-d trees and BVHs. Instead of using the object median, we use the object mean (i.e., center of mass, assuming all triangles have mass 1). I have no explanation *why* this is superior, it's just a heuristic that performed well.

3.1.5 Surface area heuristic (SAH)

This is a more advanced technique for partitioning. Although it did not achieve superior performance to the object-mean heuristic, this technique has superior time-complexity (log-linear) [1]. (The object median and object mean algorithms are both quadratic.) The main idea of the SAH is that if we assume a uniform distribution of viewing rays, the expected tree-traversal time for a viewing ray is proportional to the ratio of surface areas of the root node and its subnodes. Specifically, the expected time is

$$\mathbf{E}(T) = \text{Area}(T_L)\mathbf{E}(T_L) + \text{Area}(T_R)\mathbf{E}(T_R)$$

which is very difficult to calculate. However, if we assume that T_L and T_R are leaves, we see that

$$\mathbf{E}(T) = \text{Area}(T_L)N_L + \text{Area}(T_R)N_R$$

where N_L, N_R are the number of triangles in the left and right subtrees. The key observation is that as the partition plane slides, this expression is linear everywhere except when the partition plane meets the edge of a triangle. Therefore, this function must reach its minimum value at one of those $2N$ locations. If we consider partitioning along any of the 3 axes, we see that with a brute-force search there are only $6N$ locations to check for each tree node. I implemented this brute-force approach, and although it is technically quadratic, it was *incredibly* slow, since it was quadratic at *every node*. In [1], a log-linear procedure is derived to perform the partitioning, but implementing that is beyond the scope of this project.

3.2 Bounding Volume Hierarchies (BVHs)

BVHs are identical to k-d trees, except they partition the objects instead of the space. Specifically, instead of allowing a triangle to be in both subtrees like in k-d trees, BVHs allow the subtree AABVs to overlap. I found that my BVH implementation achieved superior performance to my k-d

tree implementation, but there could have been a number of inefficiencies in my code that caused this. Regardless, there are many interesting properties of BVHs. The main one is that BVHs can be adjusted easily and efficiently, so that dynamic scenes can be rendered [3]. Only the AABVs need to be adjusted, since changing a triangle's position in a BVH does not move it to a different subtree, unlike in k-d trees. Also, the SAH can be applied to BVH construction, although I did not attempt this.

3.3 Response to question in class: Breaking the SAH

In my demo I was asked if there was an edge-case scene that "breaks" SAH-based tree construction. The answer to that question is technically no: there is not a scene where the SAH method would produce a worse tree than, say spatial median partitioning. In constructing a counterexample, the property we would want to exploit is that the SAH optimizes viewing *in all directions equally*, so if you were only going to view a scene from one perspective, the SAH may give you a sub-optimal tree.

The difficulty of constructing a counter-example is mainly due to the fact that the SAH puts heavy emphasis on constructing balanced trees (i.e., where $N_L \approx N_R$ everywhere). And the interesting thing about k-d trees is that if you intersect the AABV of the tree anywhere, you will end up traversing all the way to a leaf, assuming the tree is unoptimized. So what really matters is the depth of the tree, which the SAH tries to minimize. However, if the tree is *optimized* (i.e., bounding volumes are reduced to fit the triangles exactly), then you may be able to construct a counter-example where the SAH fails. Since triangles can be in multiple subtrees, these optimizations were too much for me to implement.

4 Future Work

An obvious way to extend this project is to implement the log-linear SAH-based partitioning algorithm found in [1], or one of the similar-complexity, more practical variants discussed in both [1] and [2]. There are also a number of modifications and code improvements that can be made to improve the ray-tracer efficiency.

Another interesting way to extend this project would be to add reflections (recursive ray-tracing) or other asthetic improvements (e.g., translucency, anti-aliasing/sampling). Doing these calculations would be just as efficient as primary ray intersections with a BVH/k-d tree. The primary goal of this project was improving rendering performance, so little attention was paid to producing impressive or realistic images.

5 Work Used

I used code found online as a reference for my k-d tree implementation, mainly for the OOP modeling for bounding boxes and my Shape interface. Specifically, I referenced code found at blog.frogslayer.com/kd-trees-for-faster-ray-tracing-with-triangles. However, there is a major issue

with the author's implementation: it is not a k-d tree, it is actually a BVH. The author seems unaware of this error, since they still try to count the triangles in both subtrees. However, the author also used the object-mean split, which I found to be very effective for both BVHs and k-d trees.

6 References

[1]: I. Wald, V. Havran, *On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)*, Proc. 2006 IEEE Symposium on Interactive Ray Tracing, DOI 10.1109/RT.2006.280216

[2]: V. Havran, *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague, 2001

[3]: I. Wald, S. Boulos, P. Shirley, *Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies*, ACM Transactions on Graphics, January 2007, Vol. 26 No. 6