
IMPLEMENTING AND BENCHMARKING DIFFERENT ACCELERATION DATA STRUCTURES FOR RAY TRACING

MASTER PROJECT RENDERING TRACK

AUTHOR

ALHAJRAS ALGDAIRY

ADVISOR

PROF. DR.-ING. MATTHIAS TESCHNER

*Albert-Ludwigs-University of Freiburg
Chair of Computer Graphics*



MAY 27, 2022

Acknowledgments

This project results from hard work and cumulative knowledge gained through seminars, labs, and lectures in the chair of computer graphics at the University of Freiburg under Prof. Dr.-Ing. Matthias Teschner guidance and supervision. Those materials and their resources were the fundamental building blocks to reaching this point with the proper feedback from the professor.

Previous work

This project is based on a simple raytracer that I have been developing in a Lab course at the University of Freiburg. The raytracer provides basic functionality, but it is not a commercial raytracer; it still needs some improvement from different perspectives. This project will focus only on the acceleration data structures and explore different methods, and tests their performance against each other.

Abstract

This report aims to implement and analyze different acceleration data structures integrated into a raytracer. Three types will be discussed: Bounding Volume Hierarchies (BVH), Linear Bounding Volume Hierarchies (LBVH), and Kd-Tree. The report shows the proof of concept and how to implement each data structure. Furthermore, we will explore each method's strengths and weaknesses by using different scenes and models.

Keywords: [ray tracing] [acceleration structure] [BVH] [LBVH] [object subdivision] [spatial subdivision] [KD-Tree]

Contents

1	Introduction to Ray tracing	3
1.1	Overview	3
1.2	Motivation of using acceleration data structures	4
2	Bounding Volume Hierarchies (BVH)	5
2.1	Introduction	5
3	Implemntation	6
3.1	Construction	6
3.1.1	Visual Illustration	7

1 Introduction to Ray tracing

1.1 Overview

Ray tracing is an algorithm to simulate how light behaves in a 3D scene to generate real-life digital images in a computer. This process is called rendering. Rendering is used in various applications, such as Gaming, Animation, Engineering, and Moviemaking.

Raytracing has three different implementations: Forward Raytracing, Backward Raytracing, and Hybrid Raytracing; regardless of which implementation is used, the general idea of the algorithm is quite simple but extremely powerful; it is to project the 3D scene into a 2D plane (Image), to do so, a resolution must be chosen beforehand to divide the plane into small squares named (Pixels), then we try to evaluate the color and illumination of each pixel. We cast several rays from the sensor/camera towards the scene for each pixel. We search for any intersection with all primitives, and if we hit one, we save it in a list. Depending on the distance between the sensor and the hit primitive, we can evaluate the nearest one to the sensor and pick it up. We can retrieve the primitive color from its original color property and start shading it. This includes knowing if the primitive is in a dark or bright region in the scene. This expensive process is recursively executed until we hit the light source or reach a predefined depth. For thousands of primitives and more, testing the intersection is a performance challenge.

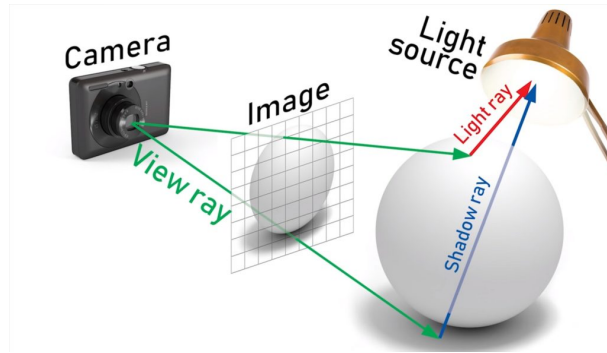


Figure 1: Ray casting illustration, where rays are travelling from the camera/sensor towards the scene.(GEORGE KIMATHI, 2020)

Other approaches such as Rasterization can be used to overcome these obstacles; however, Raytracing algorithms can provide more realistic images than Rasterization even though they take more time to render a scene. This is the reason why Rasterization is used for real-time applications such as gaming more than Raytracing; on the other hand, Raytracing is more often used in offline applications such as simulations and interior design software.



Figure 2: NVIDIA RTX Technology Realizes Dream of Real-Time Cinematic Rendering.(Brian Burke, 2018)

In Game Developers Conference - NVIDIA announced NVIDIA RTX™, a ray-tracing technology that brings real-time, cinematic-quality rendering to content creators and game developers. (Brian Burke, 2018), the results were promising and shows that having a real-time raytracer is possible with the advancement of GPUs.

The lighting and the shadows are so detailed in the scene that they look like a picture taken from a camera and not generated from a computer. The real challenge is real-time raytracers, which require rendering approximately 50 high-quality frames per second.

We should understand how challenging to render one frame that consists of thousands of primitives in less than one second.

1.2 Motivation of using acceleration data structures

In the Raytracing pipeline, we mentioned that each ray must execute an intersection test with all the primitive contained in the scene. This means if we have N number of pixels and P number of primitives, this will produce a complexity of $O(N.P)$, this means even if one pixel only contain a primitive we would have to go to all other pixels and still test them against all primitives.

Brute forcing complexity is high, increasing linearly with the number of primitives. Moreover, some features such as anti-aliasing require more rays per pixel which increases the complexity to $O(N.P.S)$ where S is number of samples per pixel.

Using more samples will often produce a higher quality frame; additionally, for better illumination results, more recursion (Shadow rays) must be cast after each intersection. We need more rays but with fewer intersection tests as possible.

Let us illustrate this with a simple example, giving a raytracer that uses S : number of samples per pixel = 5, N : number of primitives = 100000, and P : number of pixels = 1280 x 1024, maximum depth = 3. This will give us a number of intersection tests, approximately = $1280 \times 1024 \times 100000 \times 5 \times 3 = 1.96608 \times 10^{12}$ intersection test, assuming the machine we are using will spend 0.01ms in each ray. This means for this simple scene the raytracer will render the frame in 220 days. Some scenes contain millions of primitives making, rendering them by brute-forcing impossible.

Luckily one can use preprocessing algorithms to reorder and group the primitives to make them quickly tested. Spatial subdivision and object subdivision are the two basic types of data acceleration structures. Spatial subdivision algorithms partition three-dimensional space into areas and keep track of which primitives overlap which regions. On the other hand, object subdivision algorithms gradually subdivide the scene's objects into smaller groups. This way, we can only test the primitives that have a higher probability to intersect with the ray rather than testing all primitives that are not relevant to the region the ray passes through.

2 Bounding Volume Hierarchies (BVH)

2.1 Introduction

The basic idea of the BVH is simple yet powerful. It is to wrap all primitives in a virtual bounding box. This box will act as metadata to show the limits of the primitive and has no idea of how the shape of the primitive inside it. This concept will make it easier to test the primitives because one can wrap a complex model with one bounding box, and if the ray intersects the bounding box of the model, then and only then can we test its primitives.

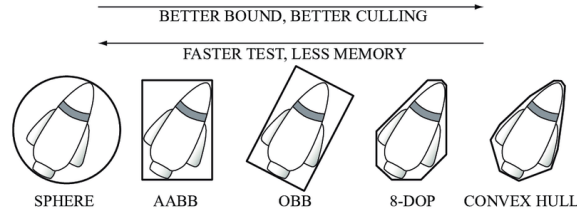


Figure 3: Bounding volumes: sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB), eight-direction discrete orientation polytope (8-DOP), convex hull (Ericson, 2004)

The bounding box can have different shapes, as shown in Figure. The simpler the bounding box is, the faster and easier to test the intersection. However, the less tightened it becomes and space between the primitive the bounding box boundaries. A compact bounding volume will assist us have the fewest overlaps with other bounding volumes in our scene, while a quick intersection test will help reduce time complexity. In this report, we will be using AABB, because it is easy to implement and easy to test its intersection, and less memory consumption because it only needs to save two double points the minimum and maximum edges of the bounding box. One can use a combination of all of them but in this report we will only use the AABB.

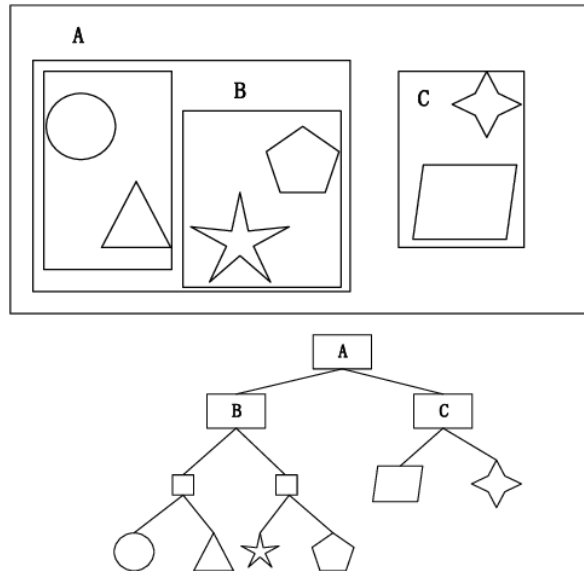


Figure 4: BVH tree result based on a simple scene. (Ericson, 2004)

Let us look at the figure as an example; this scene is consistent of 6 objects. Looking at

the generated binary tree, we can note that every two objects are bounded by one AABB, then each two AABB are bounded by a bigger AABB. We recursively do this until we reach the root node that covers the limits of the whole objects.

Without the BVH, we will have to go through all the 6 objects for each ray, even for the rays that do not intersect with any object. On the other hand, when we introduced BVH, we can note that for the rays that do not even intersect with the root AABB node, we do not go further to test its children. This means we will only test once rather than six times. Looking at the sphere, if a ray intersects with it, we will only have to go through a path that goes A -> B -> AABB sphere parent -> Sphere. These are four tests instead of 6.

We know that for a binary tree, the worse case complexity is $O(n)$, but if it is a balanced tree, it becomes $O(\log n)$, this is significant, but the catch is we should try to build a balanced binary tree. This is where splitting criteria come in handy.

3 Implemntation

BVH has two main implementation pillars, Construction of the BVH tree and Traversal over the tree.

3.1 Construction

Before going through the construction details, we have to discuss one essential key in building the BVH tree. BVH tree comes in different flavors depending on its splitting criteria. Since we are building a tree, the critical question is, when do we split the node? and which axis to choose for splitting? Firstly we will answer the first qquestion. There are three popular strategies to split the node:

- **Median of the centroid coordinates (Object median):** The median of primitives, meaning if we have the next primitive positions in the x-axis as follows 1, 3, 3, 6, 7, 8, 9 the median will be the primitive in the middle, which is 6. This strategy will produce a well-balanced tree because it splits the primitives into two equal subtrees. Because this method is intuitive to implement and produces a well-balanced tree, this strategy will be adopted in this report.
- **Mean of centroid coordinates (Object mean):** Using the man of the primitives going back the example 1, 3, 3, 6, 7, 8, 9, the mean of this set is 5.2, where the left child node will contain 1, 3, 3 and the right child 6, 7, 8, 9
- **Spatial median:** Splitting the primitives volume into two equal parts. For example, if we have four spheres with the next radius 1, 1, 1, 4, this strategy will split it as follow, the left child is 1, 1, 1 and the right child is 4 because it will split based on the volume/area of the sphere and not the position as the previous strategies.

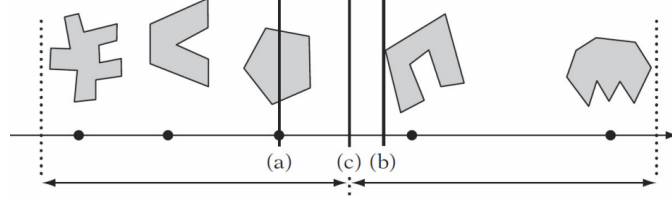


Figure 5: An example of (a) Object median splitting (b) Object mean splitting (c) Spatial median splitting based on (Ericson, 2004)

To answer the second question, which axis to choose for splitting? One can use any axis, but this is a naive way to split. What happens if all primitives have the same z-axis and y-axis, but only the x is changing, and we randomly choose to split by z-axis! This will usually produce an unbalanced tree. The optimal way is to calculate the longest axis for each node (AABB box). This way, we can try to split by the longest axis, which will produce a better-balanced tree. We can find the longest axis of the AABB box by using its diagonal d .

$$d = \text{AABB}_{max} - \text{AABB}_{min} \quad (1)$$

In the Figure, we have four spheres and an AABB covering all the spheres. If we would like to choose one axis to split from, we can randomly start with the x-axis. As it is illustrated, choosing this axis will not give us a middle point to split point since all spheres have the same x-axis. Same for the y-axis. On the other hand, we can see that the Z-axis will split the AABB box into two AABB boxes, each containing two spheres.

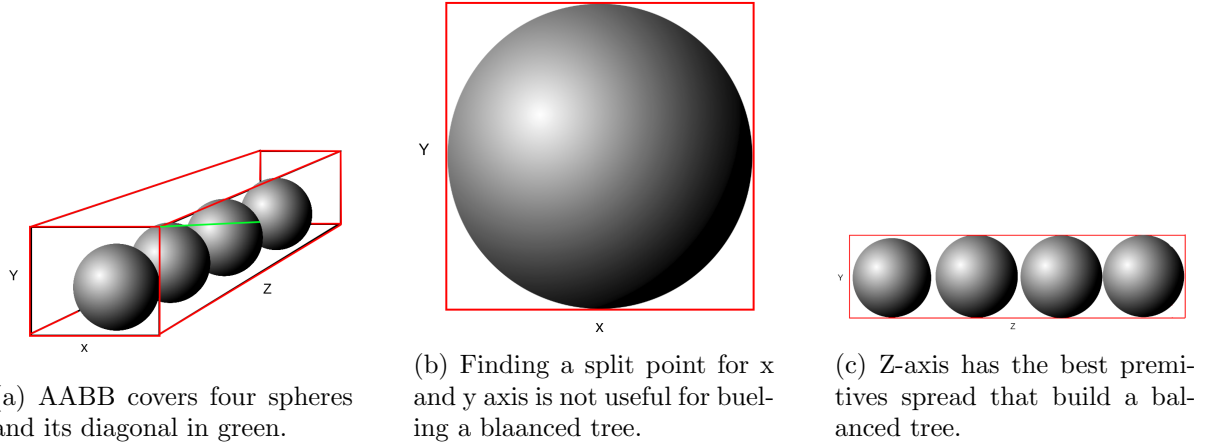


Figure 6: The scene contains four spheres with the same x-axis and y-axis but different z-axis.

3.1.1 Visual Illustration

For a better illustration of how to build the BVH tree by using the Median of the centroid coordinates splitting criteria and also by choosing the longest axis to split from, we will look at the next scene as an example:

Firstly, we start creating the root node by calculating the *minimum_edge* (the smallest bounding point in our AABB) and the *maximumu_edge* point (the largest in the AABB box). This can be easily done by iterating through all primitives and assigning the smallest primitive centroid to the *minimum_edge* and the largest primitive centroid to the *maximum_edge*.

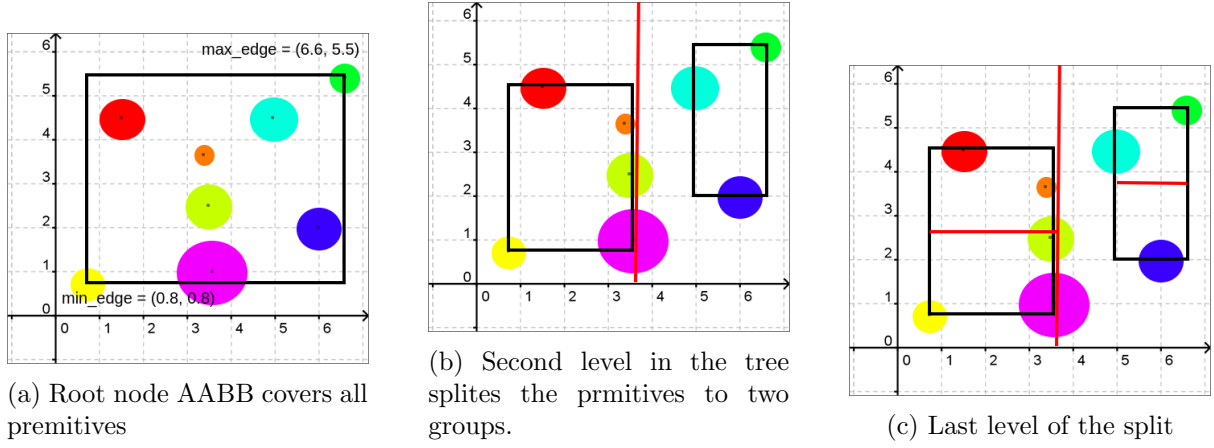


Figure 7: Creating AABBs boxes

The second step is to split the primitives in the node by using the AABB boundaries. As we mentioned, we will find the longest axis. To find the longest axis, we subtract the *minimum_edge* from *maximum_edge* and find the maximum of both axes $\max(\text{maximum_edge} - \text{minimum_edge})$. This will give us $\max((6.6 - 0.8) - (5.5 - 0.8)) = \max(5.8, 4.7) = 5.8$, which is the x-axis.

Now we calculate the splitting point by just halving the distance between *maximum_edge* and the *minimum_edge*, $\text{split_point}_x = (0.8 + 6.6)/2 = 3.6$.

We then create a left node that contains the primitives less than 3.6 and a right node that contains the primitives with a centroid > 3.6 .

Recursively we keep doing this until we reach a *maximum_leaves* parameter which is predefined, and in this report, it is 1. This means the leaf node will only hold one primitive.

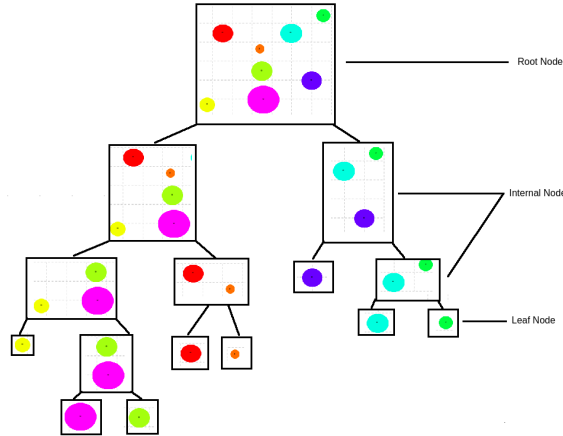


Figure 8: BVH Tree example by using the median split criteria